Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Nanut

Varno izvajanje programov v operacijskem sistemu GNU/Linux

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil XeLaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomskem delu preglejte področje omejevanja dostopa do sistemskih virov. Osredotočite se na sistem Linux in predstavite mehanizme, ki omogočajo tovrstno omejevanje. Opišite prednosti in slabosti predstavljenih mehanizmov in podajte primere uporabe. Izdelajte samostojen program, ki omogoča omejevanje dostopa do datotečnega sistema, omrežne povezave, pomnilnika in procesorja ter porabe sistemskega časa. Izdelan program uporabite kot orodje za omejeno izvajanje programov oddanih preko spletnega sodniškega sistema.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Nanut, z vpisno številko **63090157**, sem avtor diplomskega dela z naslovom:

Varno izvajanje programov v operacijskem sistemu GNU/Linux

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 22. decembra 2014 Podpis avtorja:

Zahvaljujem se prijateljici Boženi, saj brez njene vztrajne podpore ne bi zbral volje za dokončanje tega dela.

Zahvaljujem se mami in stricu, ki sta me vsak teden spomnila na pomen zaključka študija in očetu, ki je spodbujal njuno opominjanje.

Zahvaljujem se sestri, ki me je zadnje dni gnala k delu.

Zahvaljujem se tudi prijateljici Renati, ker je moje odlašanje razumela in prijatelju Vidu, ker se z njo ni strinjal.

Prav posebno pa se zahvaljujem mentorju doc. dr. Tomažu Dobravcu, tako za pomoč pri izdelavi dela kot za prenašanje moje stalne nezanesljivosti.

Kazalo

Seznam uporabljenih kratic in simbolov

`gid`	Naravno število, ki identificira posamezno skupino uporabnikov. Ime izhaja iz angl. ***g**roup **id**entifier*.
GNU	Izvedenka operacijskega sistema Unix, ki navadno uporablja jedro Linux. GNU je rekurzivna okrajšava angl. *GNU's **N**ot **U**nix*.
IPC	Medprocesna komunikacija, angl. ***I**nter **P**rocess **C**ommunication.*
`pid`	Naravno število, ki identificira posamezen proces (program v izvajanju). Ime izhaja iz angl. ***p**rocess **id**entifier*. Soroden je `ppid` (angl. ***p**arent* `pid`), ali identifikator procesovega predhodnika.
POSIX	Družina standardov organizacije IEEE za ohranjanje združljivosti med različnimi operacijskimi sistemi, običajno med različnimi izvedenkami operacijskega sistema Unix. Ime je kratica za angl. ***P**ortable **O**perating **S**ystem **I**nterface.*
Potegni in spusti	Funkcionalnost grafičnih vmesnikov, da gradnike na zaslonu s pomočjo kazalne naprave (miške ali sledilne ploščice) premaknemo na drugo mesto in s tem sprožimo različne akcije. Običajno je to premikanje ali kopiranje datotek v drug imenik, ali pa odpiranje datoteke z določenim programom. Angleški izraz za njo je *drag and drop*.
Odložišče	Včasih »sistemsko odložišče«, je prostor v delovnem pomnilniku operacijskega sistema, kjer lahko uporabniki hranijo začasne podatke. Navadno ga uporabljamo s tipkovnimi bližnjicami `CTRL+C` in `CTRL+V` oziroma z ukazi kopiraj in prilepi. V angleščini mu rečemo *clipboard*.
System V	Znan tudi kot SysV, je eden prvih komercialnih operacijskih sistemov Unix. V okviru sodobnih operacijskih sistemov se omenja takrat, ko implementirajo enake ali zelo podobne programske vmesnike njegovim. Znak »V« v imenu je rimska številka pet.
`tid`	Naravno število, ki identificira posamezno nit procesa. Ime izhaja iz angl. ***t**hread **id**entifier*.
`uid`	Naravno število, ki identificira posameznega uporabnika. Ime izhaja iz angl. ***u**ser **id**entifier*.

Povzetek

Diplomsko delo obravnava težavo varnega izvajanja neznanih programov v operacijskem sistemu GNU/Linux. Potreba po rešitvi problema se pojavi pri sodniških sistemih za programske domače naloge in pri programih za vrednotenje rešitev programerskih tekmovanj. V obeh primerih mora strežnik izvajati programe, ki jim ne more zaupati. Če bi uporabnik takšnega sistema dobil dostop do datotek ostalih uporabnikov, bi lahko njihove rešitve ukradel ali jih celo pokvaril. V okviru dela smo razvili program `gaol`, ki z uporabo mehanizmov v jedru Linux takšno varnost omogoča. Poleg tega pa smo razvili spletni sodniški sistem za izvajanje in preizkušanje programskih domač nalog, ki je grajen na programu `gaol`.

Ključne besede: varnost, lahka virtualizacija, GNU/Linux, vsebniki, sodniški sistem za programske naloge.

Abstract

This diploma thesis addresses the issue of unknown program execution in the GNU/Linux operating system. The need for such a solution is apparent in online submission systems for programming assignments and evaluation systems used in programming competitions. Untrusted programs must be executed in both cases. If a user obtains access to other users' files, they can steal and sabotage their solutions. In the context of this work, the program `gaol` has been developed. It supports this kind of security by using available mechanisms in the Linux kernel. In addition to it, an evaluation system build on top of `gaol` for executing and testing programming assignments has also been developed.

Keywords: security, lightweight virtualization, GNU/Linux, containers, evaluation system for programming assignments.

Poglavje 1

Uvod

V diplomskem delu iščemo rešitev za varno izvajanje neznanih programov v operacijskem sistemu GNU/Linux. Programi imajo privzeto dostop do večjega dela sistemskih virov brez nadzora. Uporabijo lahko ves delovni pomnilnik našega računalnika, neznancem lahko pošiljajo sporočila po spletu in zapolnijo lahko naš trdi disk.

Običajno to ni težava, saj programe prostovoljno namestimo in njihovemu delovanju zaupamo. Postane pa skrb, če želimo izvajati popolnoma neznane programe s spleta. To potrebujemo v primeru, če razvijamo sodniški sistem za domače naloge v obliki programov, ali če smo skrbniki sistema za vrednotenje rešitev s programerskih tekmovanj. V primeru, da se uporabnik izogne našim varnostnim mehanizmom, lahko skopira domače naloge in rešitve drugih uporabnikov, ali pa jih celo pokvari. Poleg tega bi bil takšen program zanimiv tudi kot orodje za izvajanje znanih zlonamernih programov, saj bi lahko brez posledic opazovali njihovo delovanje.

Rešitev bomo implementirali v operacijskem sistemu GNU/Linux zato, ker je v zadnjih letih razvoj v smeri varnejšega izvajanja programov bistveno napredoval. Pojavile so se alternative strojni virtualizaciji, ki zahtevajo manj sistemskih virov in delujejo hitreje, medtem ko nudijo podobno mero izolacije od ostalih delov operacijskega sistema.

Težavo smo rešili z razvojem programa `gaol`, ki omogoča varno izvajanje katerihkoli neznanih programov preko ukazne vrstice. S programom lahko nastavimo omejitve posameznih sistemskih virov, ki jih bodo imeli omejeni programi na voljo, iz omejenih okolij pa ne bodo mogli pobegniti. Implementirali pa smo tudi spletni sodniški sistem za domače naloge, ki `gaol` uporablja za zagotavljanje varnosti. Ker tvorijo njegovo podlago učinkoviti mehanizmi, je naloge možno reševati interaktivno kar na njem.

Poglavje 2

Mehanizmi za omejevanje izvajanja programov

V tem poglavju bomo opisali različne mehanizme operacijskega sistema GNU/Linux, ki jih lahko uporabimo za omejevanje izvajanja programov. Namen mehanizmov je, da omejijo nek del izvajanja programa na nivoju operacijskega sistema. Zaradi tega spreminjanje programa ni potrebno, mehanizmi pa so neodvisni od programskega jezika, v katerem je program napisan. Najprej pa sledi opis osnovnih delov sistema, ki bodo v poglavju pomembni.

2.1 Pomembni pojmi v operacijskem sistemu GNU/Linux

Programi in procesi
Zaradi različnega delovanja mehanizmov je potrebno razlikovati pojem *programa* od pojma *procesa*. Program je zapis programske kode v datoteki ali pomnilniku. Ko program zaženemo postane proces. Proces je skupina ene ali več niti, vsaka od njih pa lahko izvaja svoj program. Nekateri mehanizmi vplivajo na cel proces (nad skupino niti), nekateri vplivajo na točno določeno nit, nekateri pa celo na več procesov hkrati.

Uporabniki
Vsak proces pripada uporabniku. Najbolj privilegiran je uporabnik z imenom `root`, ki v operacijskih sistemih GNU/Linux predstavlja t. i. *super uporabnika*, saj lahko brez omejitev izvaja vse programe in kliče vse funkcije. Včasih mu rečemo *korenski uporabnik*, ker v angleščini *root* pomeni koren. Njegov identifikator (`uid`) je `0`. V drugih operacijskih sistemih se za podobno vlogo uporabljajo izrazi kot administrator ali skrbnik.

Lupine
Vsak uporabnik ima nastavljeno privzeto *lupino*[1], ki se zažene ob vpisu v sistem. Lupine so programi, ki implementirajo tolmačenje ukazov v ukazni vrstici. Omogočajo brskanje po prej-

[1] Podatek o privzeti lupini uporabnika se nahaja v datoteki `/etc/passwd`. Najlažje ga uredimo s programom `chsh`, ki najprej preveri, če vnesena lupina zares obstaja, saj bi sicer lahko prišli do nedelujočega stanja. Seznam vseh nameščenih lupin prebere iz datoteke `/etc/shells`.

šnjih ukazih, ki smo jih izvedli, združevanje programov s preusmerjanjem standardnih vhodov in izhodov, lažje krmarjenje skozi imenike datotečnega sistema in lažje iskanje programov, ki so nam na voljo. Primeri priljubljenih lupin so Bash (***B**ourne **A**gain **Sh**ell*), Zsh (***Z sh**ell*) in Fish (***F**riendly **i**nteractive **sh**ell*). Razlikujejo se v funkcionalnostih kot so barvanje sintakse v ukazni vrstici in dopolnjevanje uporabnikovega vnosa, vsaka pa ima svoj skriptni programski jezik.

Datotečni sistem
Pot do vsake datoteke v sistemu se začne z imenikom `/` (koren ali angl. *root*), ki vsebuje tudi posebne datoteke, ki ne predstavljajo podatkov na enem od pomnilniških medijev. Vse naprave v sistemu so predstavljene kot takšne posebne datoteke, na primer omrežne kartice, tipkovnice, miške, spletne kamere, baterije prenosnikov in podobne. Njihove datoteke lahko najdemo v imeniku `/dev`. Kot datoteke so predstavljene tudi funkcije, ki jih navzven omogoča jedro sistema. Takšne so na primer ustvarjanje nizov naključnih števil (`/dev/urandom`), nastavitve jedra (`/proc/sys/kernel`) in seznam vseh procesov s pripadajočimi podrobnostmi (`/proc`). Funkcije uporabljamo tako, da datoteke odpremo na običajen način našega programskega jezika in njihovo vsebino preberemo. Če funkcije omogočajo spreminjaje podatkov pa v njih pišemo. Zaradi tega, ker vse prikazane datoteke niso »resnične«, ampak so samo navidezne, temu rečemo navidezni datotečni sistem ali VFS (angl. ***V**irtual **F**ile **S**ystem*).

Priklopi (angl. *mounts*)
Mejnikom različnih datotečnih sistemov v drevesu VFS rečemo *priklopne točke* (angl. *mount points*). Na primer, `/` ali koren je prva priklopna točka. Pod njim se lahko nahaja `/home`, ki vsebuje podatke uporabnikov in je navadno svoja priklopna točka, saj se nahaja na drugem delu pomnilniških medijev od korena. Imenik `/proc` je še ena ločena priklopna točka, saj sploh ne vsebuje podatkov s pomnilniških medijev ampak je vmesnik do funkcij jedra, ki so predstavljene kot datoteke.

Na različnih priklopnih točkah je lahko viden en in isti priklop. S priklopi in priklopnimi točkami upravljamo s sistemskim klicem `mount` (*priklopi*), ki omogoča ustvarjanje novih in premikanje obstoječih priklopov. Z njim lahko ustvarimo navidezne datotečne sisteme kot sta tista, ki se navadno nahajata na `/proc` in `/dev`, lahko pa tudi priklopimo bločne naprave, kot so particije trdih diskov in drugih pomnilniških naprav, na priklopne točke v drevesu datotečnega sistema, kjer bodo nato dostopne njihove datoteke. Sistemski klic podpira tudi t. i. *povezovalne priklope* (angl. *bind mount*), ki omogočajo, da je vsebina enega priklopa vidna na več priklopnih točkah. Ti delujejo podobno kot simbolične povezave datotek, le da obsegajo celotno vsebino priklopa oziroma celotno poddrevo v datotečnem sistemu, ter da jih ne moremo pomotoma pokvariti (datoteko, na katero kaže simbolična povezava, lahko izbrišemo). Nasprotje sistemskega klica `mount` je sistemski klic `umount`, ki priklop s priklopne točke odstrani, za sabo pa pusti prejšnje stanje. Imenik po klicu `umount` ni nujno prazen, saj vsak klic `mount` na priklopno točko doda priklop kot na sklad, tako da lahko klice `mount` gnezdimo.

Posebni imeniki v datotečnem sistemu
Pot do datoteke, ki se začne s `/`, je absolutna pot do datoteke, saj navaja celotno pot od korena do datoteke. Relativne poti navajajo pot od trenutnega delovnega imenika procesa naprej. Vsak proces hrani podatek o trenutnem delovnem imeniku, ki ga lahko dobimo s funkcijo `getcwd` (angl. ***get c**urrent **w**orking **d**irectory*) ali ga preberemo iz navidezne datoteke `/proc/self/cwd`. Vsak proces hrani tudi povezavo do imenika, ki predstavlja njegov absoluten koren. Te povezave ne moremo dobiti s pomočjo sistemskega klica, je pa na voljo kot vsebina

navidezne datoteke `/proc/self/root`.

Imenik `/proc` navadno predstavlja navidezni datotečni sistem `procfs`. Navidezne datoteke in imeniki, ki jih vsebuje, z branjem omogočajo vpogled v podatkovne strukture jedra. Te vsebujejo podatke o centralnih procesnih enotah (`/proc/cpuinfo`), statistiko o pomnilniških napravah (`/proc/diskstats`), podatke o samem jedru: s katerimi argumenti je bilo pognano, kako je bilo prevedeno ..., seznam podprtih datotečnih sistemov (`/proc/filesystems`), podrobnosti o podsistemih omrežja (`/proc/net`) in druge. V nekatere izmed posebnih datotek lahko pišemo, da spremenimo nastavitve jedra. Mnogo se jih nahaja v imeniku `/proc/sys`, na primer datoteka `/proc/vm/drop_caches`, ki ob pisanju povzroči praznjenje sistemskih predpomnilnikov [38]. To je lahko uporabno za pripravo okolja na merjenje hitrosti programov ali razhroščevanje in preizkušanje jedra.

Za procese so v `/proc` bistveni podimeniki `/proc/PID`, kjer `PID` predstavlja identifikator posameznega procesa. Na primer, v `/proc/1` se nahajajo podatki prvega procesa na sistemu, ki mu navadno rečemo `init`[2]. Proces lahko dostopa tudi do svojih lastnih podatkov preko povezave `/proc/self`. V podimeniku se nahajajo osnovni podatki kot so ime procesa in njegov trenutni delovni imenik, najdemo pa tudi zanimivejše stvari, na primer celoten navidezni pomnilnik procesa, predstavljen kot ena datoteka `mem`, sklad procesa v datoteki `stack` ali podrobnosti trenutnega sistemskega klica, ki ga proces izvaja, v datoteki `syscall`. Imenik vsebuje tudi podimenik s podrobnostmi o posameznih nitih procesa (`/proc/PID/task`) in podimenik s seznamom odprtih datotečnih deskriptorjev (`/proc/PID/fd`).

Datotečni deskriptorji in cevi

Najpogosteje se srečujemo z datotečnimi deskriptorji 0, 1 in 2, ki v tem zaporedju predstavljajo standardni vhod (`stdin`), standardni izhod (`stdout`) in standardni izhod za napake (`stderr`). Ko poganjamo programe v lupini z njimi običajno komuniciramo preko teh deskriptorjev. Vnos s tipkovnice lahko zamenjamo s preusmeritvijo iz datoteke v programov standardni vhod, podobno pa lahko storimo tudi z izhodi, da program rezultate zapisuje v datoteko namesto na ekran terminala ali pa jih posreduje drugemu programu. Te preusmeritve so implementirane s pomočjo cevi (angl. *pipe*), ki tvorijo enosmeren kanal z vhodom in izhodom, preko katerega lahko dva procesa komunicirata. Takšne cevi imenujemo anonimne, ustvarimo pa jih s funkcijo `pipe`. Imenovane cevi (angl. *named pipe*) delujejo enako, s to razliko, da za njih obstaja datoteka v datotečnem sistemu. Ustvarimo jih s funkcijo `mkfifo` ali s sistemskim klicem `mknod`.

Signali

Komunikacija med procesi poteka tudi v obliki signalov, ki so preprosta eno številčna sporočila. Uporabljajo se na primer v lupini, ko želimo izvajanje procesa prekiniti s pritiskom kombinacije `CTRL+C`. Takrat se tekočemu procesu pošlje signal `SIGINT` (*interrupt*). Podobno je pri zaključevanju procesa preko grafičnega vmesnika s klikom na `X`, ki običajno povzroči pošiljanje signala `SIGTERM` (*terminate*). Za večino signalov lahko proces definira rutino za ravnanje s signalom (angl. *signal handler*). Če prejme signal, za katerega rutine ni definiral, operacijski sistem proces prekine. Izjema sta signala `SIGKILL` in `SIGSTOP`, saj za njiju ni mogoče definirati rutine za ravnanje. `SIGKILL` proces vedno nemudoma prekine, `SIGSTOP` pa izvajanje procesa zamrzne dokler ta ne prejme signala `SIGCONT`.

[2] Čeprav je `init` tradicionalno prvi program, ki ga operacijski sistem ob zagonu požene, ga je v mnogih distribucijah zamenjal novejši in kompleksnejši `systemd`. Beseda `init` je sicer vedno predstavljala različne programe, ki pa so imeli enako ime in enako vlogo.

2.2 Mehanizem `chroot`

Eden najenostavnejših mehanizmov za omejevanje izvajanja programov je `chroot`. Sam izraz je krajšava angleškega *change root*, ki dobesedno pomeni spremembo korena. V okviru operacijskega sistema GNU/Linux to pomeni spremembo korenskega imenika v navideznem datotečnem sistemu.

Uporabljamo ga lahko na dva načina. Prvi je pomožen program z imenom `chroot` [7], s katerim lahko v lupini določimo nov korenski imenik in ime programa, ki se bo v njem izvedel. Drugi je istoimenska funkcija iz osnovnega nabora knjižnic [26], ki spremeni korenski imenik trenutno izvajajočemu programu. Oba načina sta v osnovi implementirana s pomočjo sistemskega klica, ki nosi isto ime. V vseh primerih so za izvedbo klica potrebne pravice super uporabnika. Razlog za to omejitev je varnost, podrobnosti pa bodo opisane kasneje.

Mehanizem deluje tako, da spremeni tolmačenje poti do datotek. Če program pokliče `chroot` z argumentom `/nov_koren`, se bodo odslej vse absolutne poti začele v tem imeniku in ne v pravem korenu. Na primer, če želi program odpreti datoteko s potjo `/etc/shadow`, v kateri se nahajajo zgoščene vrednosti gesel uporabnikov, mu bo operacijski sistem ponudil datoteko `/nov_koren/etc/shadow`. Posledica tega je, da program več ne bo mogel dostopati do datotek, ki so dostopne samo na poti od pravega korena naprej.

Za uporabo `chroota` je potrebno pripraviti nov korenski imenik, ki vsebuje vse datoteke potrebne za izvajanje programa. Skoraj vsak program za izvajanje potrebuje dinamični povezovalnik (angl. *linker*) in skupno standardno knjižnico programskega jezika C (`libc`). V korenskem imeniku pa potrebujemo tudi kodo samega programa. Če program že teče, je že naložen in povezan, zato lahko uporabi `chroot` na praznem imeniku, vendar to v tem primeru stori prostovoljno.

Če želimo `chroot` uporabljati kot varnostni mehanizem, ki bi programom onemogočil dostop do vseh datotek zunaj novega korenskega imenika, je potrebno paziti na njegove pomanjkljivosti.

Prva pomanjkljivost je, da branje vseh že odprtih datotek, ki jih je program odprl pred klicem `chroot`, ni onemogočeno. Imen teh datotek ne potrebuje, saj lahko do vseh svojih odprtih datotek dostopa preko datotečnega sistema `procfs` v imeniku `/proc/self/fd`. To je lahko težavno v primeru, če je naš program strežnik, ki `chroot` uporablja za obravnavanje posameznih zahtev uporabnikov v obliki ločenih podprogramov. Zahteve lahko obsegajo na primer ogled spletne strani ali pridobivanje izidov iskanja. Če napadalec dobi nadzor nad takim delom, lahko prebere vsebino vseh datotek, ki jih je strežnik uporabljal. To vključuje datoteke z osebnimi podatki uporabnikov, zasebne kriptografske ključe in datoteke s konfiguracijo, ki jih uporabi za odkrivanje dodatnih ranljivosti. Strežnik se lahko težavi izogne tako, da zapre vse odvečne odprte datoteke preden izvede podprogram za obravnavanje zahtev. Vseh datotek ne sme zapreti, saj nekatere potrebuje za komunikacijo s samim podprogramom. Značilna takšna datoteka je tista z datotečnim deskriptorjem 1 (standardni izhod).

Druga pomanjkljivost pride do izraza, če ima program znotraj omejenega okolja pravice super uporabnika, saj lahko iz okolja pobegne. To je izvedljivo zato, ker se klici `chroot` ne gnezdijo, temveč se njihovo stanje hrani kot ena spremenljivka. Če program klic `chroot` izvede drugič se imenik, ki predstavlja njegov koren datotečnega sistema, ponovno spremeni. Ker pravega korenskega imenika ne more odpreti zaradi prvega klica `chroot`, je potreben trik, ki

se izvede na naslednji način.

1. Napadalec si shrani datotečni deskriptor do trenutnega korenskega imenika `A`.
2. Ustvari nov imenik `B` in se vanj premakne s `chroot`.
3. Z uporabo shranjenega deskriptorja se premakne nazaj[3] v imenik `A`. Sedaj se njegov trenutni delovni imenik nahaja zunaj `chroot`ovega korenskega imenika.
4. Od tukaj se premika navzgor po povezavah do višjega imenika v drevesu (s `chdir ..`) dokler ne najde pravega korena.
5. Ponovno pokliče `chroot` nad pravim korenom. Sedaj je vzpostavljeno stanje pred prvim klicem `chroot`.

Za izvedbo napada je nujno, da program pridobi pravice super uporabnika, sicer ponovnega klica `chroot` ne more izvesti. Zaradi tega mora vsak program, ki želi klic izvesti varno, spremeniti lastnika trenutnega procesa v neprivilegiranega uporabnika. To se najbolj zanesljivo stori s klicem `setresuid`, ki nastavi realni, efektivni in shranjeni `uid` na določeno vrednost, saj deluje enako v različnih izvedenkah operacijskega sistema Unix. Enako je potrebno narediti za skupino, ki ji proces pripada. To se stori s podobnim klicem `setresgid`, ki nastavi enako trojico za `gid`.

Če neprivilegiran proces ob zagonu nima pravic super uporabnika, si jih lahko še vedno pridobi. To je mogoče z uporabo zastavice `setuid` na datotekah v datotečnem sistemu. Zastavica pomeni »nastavi identifikator uporabnika ob izvedbi« (angl. ***set user identifier upon execution***). Njen izid je to, da se bo program s to zastavico izvedel v imenu in s pravicami uporabnika, ki je lastnik datoteke. Če je lastnik datoteke super uporabnik, bo imel program vse njegove pravice. Ta možnost obstaja zaradi programov kot sta `su` (angl. ***substitute user***, »nadomesti uporabnika«) in `sudo` (angl. ***substitute user do***, »nadomesti uporabnika in stori«), ki omogočata izvajanje programov v imenu drugih uporabnikov, najpogosteje kot super uporabnik. Brez njiju neprivilegirani uporabniki ne bi mogli opravljati skrbniških opravil. Zaradi nevarnosti višanja pravic sta programa dolžna skrbno preveriti pravice uporabnika, ki ju poskuša izvesti. Prav tako morata biti previdno napisana, da ne pride do zlorab. `su` za dovoljenje izvajanja preverja, ali trenutni uporabnik pozna geslo uporabnika, v imenu katerega želi izvajati programe, medtem ko `sudo` uporablja bolj zapleten mehanizem, ki se nastavi v datoteki `/etc/sudoers` [42].

Še en znan program, ki uporablja zastavico `setuid`, je program `mount` [35], ki je samo vmesnik za sistemski klic z istim imenom. Za razliko od prejšnjih dveh programov `mount` ne omogoča izvajanja programov v imenu drugih uporabnikov, ampak nudi neprivilegiranim uporabnikom manj funkcionalnosti kot super uporabniku, za svoje notranje delovanje pa vedno potrebuje njegove pravice.

Ko skrbnik pripravlja okolje za `chroot` je najbolje, da programov z zastavico `setuid` sploh ne vključi v nov korenski imenik, saj bi katerikoli izmed njih lahko vseboval varnostno ranljivost. Pozoren mora biti tudi na posodobitve, ki jih mora namestiti ročno, saj orodja za samodejno posodobitev distribucije ne vključujejo novega korenskega imenika. Če posodobitev ne namesti, lahko napadalec zlorabi neodpravljene ranljivosti in pridobi nadzor nad sistemom.

Kot je bilo prej omenjeno, sistemskega klica `chroot` neprivilegiran uporabnik ne sme

[3] To omogoča funkcija `fchdir`, ki deluje enako kot `chdir`, le da kot parameter sprejme datotečni deskriptor namesto poti.

izvesti. Razlog za to je, da bi tako zlahka pridobil pravice super uporabnika in s prej opisanim postopkom pobegnil iz umetnega korenskega imenika. Izvedba tega poteka na naslednji način.

1. Napadalec ustvari imenik `A`, ki se bo uporabil za nov koren. Tega mora napolniti z obveznimi sistemskimi datotekami in programi za delovanje sistema, sicer znotraj njega ne more izvajati programov.
2. V novem imeniku ustvari datoteki `/etc/passwd` in `/etc/shadow`, ki sicer vsebujeta podatke vseh uporabnikov na sistemu in njihova gesla. Napolni ju tako, da pozna geslo super uporabnika.
3. Izvede `chroot` na novem imeniku in se vanj premakne.
4. Znotraj novega korenskega imenika napadalec uporabi program `su` in postane super uporabnik. Ker se `su` v tem primeru izvaja znotraj okolja `chroot` bo namesto sistemskih datotek `/etc/passwd` in `/etc/shadow` podatke bral iz prirejenih različic v podimeniku `A` in spremembo uporabnika odobril.
5. Napadalec sedaj uporabi prej opisan postopek da izstopi iz okolja `chroot` in pristane v pravem korenskem imeniku z vsemi pravicami super uporabnika.

Operacijski sistem FreeBSD ima nadgrajeno različico funkcije `chroot` [8]. Od ostalih se razlikuje tako, da program prekine, če odkrije shranjen datotečni deskriptor za nek drug imenik. Na ta način se popolnoma prepreči način za izstop iz `chroota` opisan v tem poglavju.

Če želimo v okolju `chroot` izvesti poljuben program neprivilegirano to ni trivialno, saj je vstop v okolje prostovoljen in bi ga program moral opraviti sam, prav tako pa bi moral sam opustiti pravice super uporabnika. To dosežemo s pomožnim programom, ki mora opraviti naslednje korake.

1. Pomožni program, ki ga zažene super uporabnik, ustvari novo nit z uporabo zastavice `CLONE_FS`. Učinek te je, da si z novo nitjo deli vse lastnosti datotečnega sistema. Te vključujejo trenutni delovni imenik, koren datotečnega sistema in masko za nastavitev dovoljenj novih datotek (`umask` [25]).
2. Nova nit, ki še ne vsebuje kode za neprivilegiran proces, mora najprej poklicati `setresuid` in `setresgid` s podatki neprivilegiranega uporabnika da odstrani svoje privilegirane pravice. Nato izvede sistemski klic `execve` da zamenja svojo kodo s kodo za neprivilegiran proces.
3. Glavna nit pomožnega programa sedaj izvede `chroot`. Ker si niti delita datotečni sistem ta sprememba vpliva tudi na neprivilegirano nit.

V zgornjem postopku potrebujeta niti nek način komuniciranja, saj morata za posamezne korake ena na drugo čakati. Če je koda za neprivilegiran proces naša, lahko to implementiramo prostovoljno, na primer preko pisanja v vnaprej dogovorjeno cev. Tako deluje pomožni program `setuid-sandbox`[4] [45]. V nasprotnem primeru je to težje izvesti, je pa mogoče z uporabo sistemskega klica `ptrace` (razdelek 2.6).

Samostojno `chroot` pogosto srečamo v uporabi, ki ni namenjena varnosti. Tak primer je popravilo poškodovane namestitve operacijskega sistema. Do tega lahko pride zaradi nedelujočih posodobitev, napačno nastavljenih nastavitvenih datotek ali namestitve drugega opera-

[4] Brskalnik Chromium, ki je svobodna in odprtokodna osnova Googlovega brskalnika Chrome, ga uporablja za osnovo pomožnega programa za poganjanje vtičnikov in svojih podprocesov.

cijskega sistema, ki spremeni ali celo zamenja zagonski nalagalnik (angl. *boot loader*). Popravila se lotimo tako, da zaženemo drug primerek operacijskega sistema GNU/Linux z drugega pomnilniškega medija in izvedemo `chroot` na particiji, ki vsebuje poškodovan sistem. Na ta način lahko nedelujoč sistem od znotraj popravimo, saj programi za nastavitev zagonskega nalagalnika vplivajo na datoteke nedelujočega sistema.

Podoben način uporabe se pojavi pri namestitvi distribucije GNU/Linux Arch Linux. Po tem, ko se skopira osnovni del sistema na mesto, ki smo ga določili, uporabimo `chroot` da stopimo vanj. Nato osnovni del ročno nadgradimo z nastavitvijo nastavitvenih datotek in namestitvijo ostalih ne osnovnih programov, da dobimo željen sistem.

Izven namena varnosti se `chroot` uporablja tudi za preizkušanje gradnje paketov programske opreme. Vsak paket vsebuje seznam programov in knjižnic od katerih je odvisen in brez katerih ne deluje. Za posamezen paket lahko ustvarimo nov korenski imenik, ki vsebuje samo te programe, in ga s pomočjo `chroota` poskušamo zgraditi v okrnjenem okolju. Če se program ne zgradi uspešno, je bil seznam odvisnosti napačen in ga je potrebno posodobiti. Na enak način lahko preizkušamo tudi obnašanje različnih različic istega programa ali knjižnice.

`chroot` ne omejuje drugih sistemskih virov, kot sta čas in pomnilnik. Pri njegovi uporabi pa lahko skrbnik hitro naredi napako, in s tem povzroči prevzem sistema s strani napadalca. Iz teh razlogov ni primeren kot splošen mehanizem za omejevanje programov. Uporabimo pa ga lahko kot kos večje rešitve, kot bo vidno v naslednjih poglavjih.

2.3 Imenski prostori

Imenski prostori (angl. *namespaces*) obsegajo šest sorodnih funkcionalnosti jedra Linux, ki omogočajo ločevanje globalnih sistemskih virov pred procesi. Če se en proces nahaja v drugem imenskem prostoru, ne more videti virov procesov iz ostalih imenskih prostorov. Glavni razlog za njihovo zasnovo je implementacija vsebnikov (angl. *container*) za lahko virtualizacijo. Pri običajni strojni virtualizaciji simuliramo celoten računalnik, pri lahki pa med skupinami procesov izoliramo samo sistemske vire. Navadno tak pristop temelji na tem, da vsi programi iz več navideznih okolij tečejo na istem jedru, imajo pa ločen pogled na razpoložljive sistemske vire. Prednosti takšnega pristopa so enostavnejša konfiguracija in pa predvsem hitrost, saj v tem primeru virtualizacija sploh ne upočasnjuje programov. Slabost je nezmožnost poganjanja programov, ki so bili napisani za drugo različico jedra ali drugo strojno arhitekturo.

Vsi procesi so privzeto del imenskih prostorov svojih prednikov, lahko pa se premaknejo v drug imenski prostor ali ustvarijo novega. Prvi način za ustvaritev novega imenskega prostora je s sistemskim klicem `clone` [32], ki je namenjen ustvarjanju novih procesov in niti. Je zelo splošen, saj so z njim implementirani tako novi procesi (v funkciji `fork`) kot niti (v knjižnici za sočasno programiranje `pthread`). Omogoča pa tudi ustvarjanje novih imenskih prostorov za nov ustvarjen proces. Drugi način je s pomočjo sistemskega klica `unshare` [30], ki ne ustvari novega procesa ampak ustvari nov imenski prostor za tekočega. Za premik procesa v že obstoječ imenski prostor je na voljo sistemski klic `setns` [29].

Katere imenske prostore želimo vključiti določimo z uporabo eno ali več od naslednjih zastavic: `CLONE_NEWPID`, `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWNS`, `CLONE_NEWUTS` ali `CLONE_NEWUSER`. Če želimo posamezen prostor samo zamenjati s `setns`, potrebujemo zanj datotečni deskriptor. Datotečni deskriptorji imenskih prostorov nekega procesa se nahajajo v njegovem `/proc` podimeniku, natančno `/proc/PID/ns`. V imeniku se nahaja po ena datoteka za vsako vrsto imenskih prostorov: `ipc`, `mnt`, `net`, `pid`, `uts` in `user`. Če želimo določen imenski prostor

ohraniti tudi po tem, ko se vsi procesi v njem zaključijo, lahko z eno od teh datotek ustvarimo povezovalni priklop nekam drugam v drevo datotečnega sistema. Enak učinek dosežemo, če katerikoli tekoč proces hrani datotečni deskriptor za eno od teh datotek. Podrobnosti posameznih imenskih prostorov bodo opisane za vsakega posebej, saj se bistveno razlikujejo med seboj.

Za uporabo imenskih prostorov mora imeti program pravice super uporabnika. Izjema je le `CLONE_NEWUSER` za imenske prostore uporabnikov, saj je ravno njihov namen ustvariti navidezno privilegirano okolje neprivilegiranim uporabnikom oziroma procesom.

2.3.1 Identifikatorji procesov

Imenski prostori identifikatorjev procesov oziroma imenski prostori PID (angl. *PID namespaces*) omogočajo, da v sistemu ločimo več množic številk `pid`. Enaka številka `pid` se lahko v različnih imenskih prostorih PID več krat pojavi. Vsak prostor vsebuje proces s `pid` 1, ki mu tradicionalno rečemo `init`. Procesi iz različnih prostorov se med seboj ne vidijo in zato z drugimi procesi ne morejo neposredno komunicirati (na primer s signali).

Ob zagonu sistema obstaja samo en imenski prostor PID. Novi prostori, ki jih ustvarimo, so njegovi potomci, ti pa lahko imajo tudi svoje potomce. Znotraj imenskega prostora PID ima prvi proces `pid` 1 in prevzame vse naloge procesa `init`. To pomeni, da samodejno posvoji vse osirotele procese, pomeni pa tudi, da mu noben od potomcev ne more pošiljati katerihkoli signalov, saj lahko `init` sprejme samo signale, za katere izrecno definira rutino za ravnanje[5], tudi če mu jih pošlje super uporabnik. Izjema so procesi iz imenskih prostorov PID njegovih prednikov, ki mu lahko pošljejo signala `KILL` in `STOP`, da imajo možnost ustaviti prostore svojih potomcev. Ko se proces s `pid` 1 v imenskem prostoru PID zaključi, jedro ostalim procesom iz istega prostora in prostorov njegovih potomcev pošlje signal `KILL` in jih s tem prekine.

Procesi lahko nemoteno komunicirajo z drugimi procesi iz istega imenskega prostora PID, lahko pa enosmerno komunicirajo tudi s procesi iz prostorov potomcev. Vsakemu procesu pripada `pid` tudi v prostoru prednikov, vendar je ta drugačen od tistega v njegovem prostoru. Preko tega `pid` lahko procesi zunaj njegovega prostora njemu pošiljajo signale. Ena posledica tega je, da ima čisto vsak proces dodeljen `pid` v prostorih vseh prednikov, ne glede na to, kako globoko v hierarhiji se nahaja njegov prostor. To pomeni, da lahko vse procese najdemo v korenskem imenskem prostoru PID.

Čeprav ima prvi proces znotraj novega imenskega prostora PID prednika, bo `pid` njegovega prednika (`ppid`) enak 0, kar pomeni, da z njegovega vidika prednika nima. To je tudi smiselno, saj njegov prednik ne sodi v isti prostor in proces z njim ne more komunicirati. Poseben primer tukaj je pravi `init`, tisti, ki je resnično prvi proces na sistemu. Njegov `ppid` je tudi 0, vendar zato, ker prednika resnično nima.

Nov imenski prostor PID lahko ustvarimo na dva načina. Če ga želimo ustvariti hkrati z novim procesom v njem to storimo s sistemskim klicem `clone` in zastavico `CLONE_NEWPID`. Ta proces bo v novem prostoru predstavljal `init`. Drugi način za ustvarjanje novega imenskega prostora PID je sistemski klic `unshare` z isto zastavico `CLONE_NEWPID`, ki pa ne deluje na tekoč proces temveč na njegove potomce. Imenski prostor PID in `pid` tekočega procesa ostaneta nespremenjena, novi potomci pa bodo ustvarjeni v novem prostoru s `pid` od 1 naprej. Premik procesa v drug imenski prostor PID s sistemskim klicem `setns` deluje enako kot `unshare`, se

[5] Običajni procesi (tisti s `pid` različnim od 1) so nemudoma prekinjeni, ko prejmejo signal za katerega niso definirali rutine za ravnanje.

pravi da učinkuje samo na potomce procesa. Datotečni deskriptor, ki ga za `setns` potrebujemo, se nahaj v datoteki `/proc/PID/ns/pid`.

Razlog za drugačno delovanje `unshare` in `setns` je to, da veliko programov ne pričakuje spremembe `pid` med delovanjem in bi zaradi tega delovali narobe. Knjižnica glibc, ki je najpogostejša implementacija standardne knjižnice jezika C v operacijskih sistemih GNU/Linux, si vrednost `pid` po prvem klicu funkcije `getpid` shrani v globalno spremenljivko, tako da sistemskega klica `getpid` nikoli več ne pokliče[6]. Če bi se procesov `pid` spremenil, tega program napisan v višjem programskem jeziku (tudi C) sploh ne bi mogel ugotoviti.

Ko se zaključi zadnji proces v imenskem prostoru PID, jedro prostor uniči. Če poskusimo uporabiti mehanizem za ohranitev prostora brez procesov, kot je to pri drugih prostorih mogoče z uporabo povezovalnih priklopov, na prvi pogled deluje. Povezava se ustvari in prostor živi. Vendar po poskusu premika novega procesa v ta prostor dobimo napako, saj imenski prostor PID ne more delovati brez procesa z vrednostjo `pid` 1.

Pri imenskih prostorih PID moramo biti posebej pozorni na imenik `/proc`, saj njegovo stanje ni odvisno od prostora tekočega procesa, temveč ostane njegova vsebina nespremenjena tudi po spremembi imenskega prostora PID [13]. Čeprav je bilo prej rečeno, da proces v ločenem prostoru ostalih procesov ne more videti, lahko preko starega imenika `/proc` prebere njihove identifikatorje in druge podrobnosti. Trditev, da jih ne more videti, kljub temu na nek način drži, saj so ti `pid` zanj čisto brez pomena. Če jih poskusi uporabljati bo dobil napako, da ne obstajajo, oziroma bo z njihovo uporabo lahko vplival na drug proces v svojem imenskem prostoru PID, ki ima slučajno enak `pid`. Takšno obnašanje lahko privede do napak, saj programi, kot so `ps`[7], `/proc` uporabljajo za pridobivanje podatkov. To težavo lahko rešimo z uporabo imenskih prostorov priklopov (razdelek 2.3.4), kot se tudi običajno počne, in na novo priklopimo imenik `/proc` s sistemskim klicem `mount`. Zanimivo je, da lahko proces, če te pomanjkljivosti ne odpravimo, z branjem simbolične povezave[8] v `/proc/self` ugotovi svoj `pid` iz prostora prednikov, ga pa ne more smiselno uporabiti.

Imenske prostore PID lahko uporabljamo preventivno, če naš program ali podprogram ne bo potreboval pošiljanja signalov. Večjo uporabnost v okviru diplomskega dela pa predstavlja lahka virtualizacija, saj so ti imenski prostori prvi korak do tega, da znotraj svojega sistema poženemo še en, navidezen operacijski sistem. Ker se števila `pid` začnejo z 1, lahko v kombinaciji z drugimi mehanizmi še enkrat zaženemo pravi `init`[9] in s tem neodvisen primerek operacijskega sistema.

2.3.2 Medprocesna komunikacija

Imenski prostori za medprocesno komunikacijo, ali imenski prostori IPC (angl. *IPC namespaces*) so imenski prostori, ki med procesi iz različnih prostorov ločijo vire za medprocesno komunikacijo. Ločijo tiste vire, ki za svoje delovanje ne uporabljajo poti v datotečnem sistemu, izvzeti pa so tudi signali, saj jih obravnavajo že imenski prostori procesov (razdelek 2.3.1).

[6] To lahko enostavno preizkusimo s pomočjo programa `strace`, ki izpisuje sistemske klice drugih programov, in enostavnega testnega programa, ki več krat pokliče `getpid`.

[7] Program `ps` omogoča vpogled v vse tekoče procese na sistemu, oziroma tiste, ki so vidni v imeniku `/proc`.

[8] Simbolične povezave lahko preberemo s sistemskim klicem `readlink`, obstaja pa tudi pomožni program z istim imenom.

[9] Da lahko `init` deluje znotraj lahko virtualiziranega sistema mora biti za to napisan, saj mora preskočiti pripravo nekaterih datotečnih sistemov in inicializacijo naprav. Sodobne izvedenke programa to podpirajo.

Mehanizmi, ki jih imenski prostori IPC podpirajo, so sporočilne vrste System V (angl. *message queues*), množice semaforjev System V (angl. *semaphore sets*) in skupni pomnilniški segmenti System V (angl. *shared memory segments*). Podprte pa so tudi sporočilne vrste POSIX.

Imenik `/proc` vsebuje podatke za uporabo z mehanizmi za medprocesno komunikacijo, njegova obravnava pa je drugačna od tiste pri imenskih prostorih procesov, saj imenika v novem imenskem prostoru IPC ni potrebno ponovno priklopiti s sistemskim klicem `mount`. Takoj ob vstopu v nov imenski prostor bo imel proces svojo kopijo teh imenikov in datotek, ki vsebujejo naslednje.

1. Imenik `/proc/sys/fs/mqueue`, ki vsebuje podatke o vseh sporočilnih vrstah POSIX na sistemu. To so na primer `queues_max` (največje število vrst za neprivilegirane procese), `msg_max` (največje število sporočil v posamezni vrsti) in `msgsize_max` (največja velikost posameznega sporočila v bajtih) [36].
2. Imenik `/proc/sysvipc`, ki vsebuje navidezne datoteke, iz katerih lahko v obliki tekstovnih tabel preberemo podatke o medprocesni komunikaciji System V na sistemu. To so na primer datoteka `msg` za sporočilne vrste, datoteka `sem` za množice semaforjev in datoteka `shm` za skupne pomnilniške segmente [39].
3. Vsebina imenika `/proc/sys/kernel`, ki je relevantna za medprocesno komunikacijo System V. To so datoteke `msgmax`, `msgmnb` in `msgmni` za sporočilne vrste, datoteka `sem` za množice semaforjev in datoteke `shmall`, `shmmax`, `shmmni` in `shm_rmid_forced` za skupne pomnilniške segmente. Pisanje v te datoteke omogoča spremembo nastavitev medprocesne komunikacije System V [38].

Kopije virov za medprocesno komunikacijo imenskega prostora IPC se izbrišejo, ko se zaključi zadnji proces v prostoru. To lahko preprečimo enako kot pri drugih imenskih prostorih, z uporabo povezovalnega priklopa.

Imenski prostori IPC so podobno kot ostali sestavina za lahko virtualizacijo, zunaj tega pa je njihova uporaba samo preventivne narave.

2.3.3 Omrežja

Imenski prostori omrežij (angl. *network namespaces*) so namenjeni ločevanju podsistemov omrežij med procesi iz različnih imenskih prostorov. To vključuje omrežne vmesnike, sklade protokolov standardov IPv4 in IPv6, požarne zidove, usmerjevalne tabele IP naslovov, imenika `/proc/net` in `/sys/class/net`, številke vrat oziroma priključke (angl. *sockets*) in tako dalje.

Fizična omrežna naprava lahko obstaja samo v enem imenskem prostoru. Ker ima računalnik omejeno število teh, se za povezovanje omrežij znotraj imenskih prostorov in tistimi zunaj uporabljajo navidezne naprave. Najenostavnejša med temi je navidezni ethernet ali `veth` (angl. ***v**irtual **eth**ernet*).

Brez dodatnega nastavljanja bo proces znotraj novega imenskega prostora omrežij imel na voljo samo eno omrežno napravo, to je naprava s povratno povezavo (angl. *loopback device*), ki ves prejet omrežni promet samo posreduje sama sebi.

Če se imenski prostor omrežij izbriše ko še vsebuje fizične omrežne naprave, se te naprave premaknejo v prvi imenski prostor omrežij na sistemu. Ne premaknejo se v prostor njegovega najbližjega prednika, tudi če ta obstaja. Podobno kot pri ostalih imenskih prostorih lahko uporabimo povezovalni priklop da to preprečimo in imenski prostor obdržimo po tem,

ko več nima procesov. To uporablja program `ip`[10], ki poleg svojih ostalih funkcionalnosti za upravljanje z omrežnimi napravami podpira tudi upravljanje z imenskimi prostori omrežij. Njegovi povezovalni priklopi se nahajajo v imeniku `/run/netns` kot posamezne datoteke z imenom ustvarjenega imenskega prostora.

Ker je upravljanje z omrežnimi napravami sistemske narave, in ker lahko posamezna fizična naprava nastopa samo v enem imenskem prostoru, bolj zapleteno uporabo imenskih prostorov omrežij običajno spremlja ravno program `ip`, s podukazom `netns`. Z njim lahko nove imenske prostore omrežij ustvarimo in v njih izvajamo programe. Med programe sodi tudi sam program `ip`, da lahko nastavimo omrežne naprave znotraj imenskega prostora. Ustvarjanje novih imenskih prostorov in premikanje med njimi je implementirano z uporabo sistemskih klicev, ki smo jih spoznali že prej: `clone`, `unshare` in `setns`, za imenske prostore omrežij pa jim moramo podati zastavico `CLONE_NEWNET`.

Imenski prostori omrežij so že brez nastavljanja uporabni, saj programom onemogočijo komunikacijo preko omrežij, kljub temu pa sledi primer povezave zunanjega imenskega prostora z notranjim s pomočjo programa `ip` in navideznih naprav tipa `veth`, saj lahko tako občutimo delovanje teh imenskih prostorov. Za vse ukaze potrebujemo pravice super uporabnika, zato je na začetku nekaterih vrstic znak `#`, ki tradicionalno pomeni, da ukaz izvaja super uporabnik. Znak `$` na začetku vrstice pomeni, da ukaz izvaja neprivilegiran uporabnik.

1. Najprej potrebujemo dve navidezni napravi tipa `veth`, eno za trenutni imenski prostor in eno za novega. Imenovali ju bomo `veth-zunaj` in `veth-noter`. Navidezne naprave tega tipa vedno ustvarimo v parih, saj so namenjene povezovanju različnih omrežij. Zato za obe potrebujemo samo en ukaz:

   ```
   # ip link add veth-zunaj type veth peer name veth-noter
   ```

2. Nato ustvarimo nov imenski prostor omrežij z imenom `ns-testni` z naslednjima ukazoma:

   ```
   # ip netns add ns-testni
   # ip netns exec ns-testni ip link set dev lo up
   ```

   V novih imenskih prostorih omrežij so sprva vse omrežne naprave izključene, zato je dodan drugi ukaz, ki vključi napravo s povratno povezavo (*loopback* oziroma *lo*) znotraj imenskega prostora. Veliko programov pričakuje, da je ta na voljo, zato jo je dobro vključiti.

3. Sedaj lahko eno od navideznih naprav premaknemo v nov prostor, obema pa je nato potrebno nastaviti naslove IP. Notranji napravi moramo naslov nastaviti znotraj imenskega prostora, saj zunaj ne bo več vidna, prav tako pa jo moramo vključiti, saj se ob premiku v nov prostor izključi. Naslova si izmislimo: `10.0.0.1` za zunanjo napravo in `10.0.0.2` za notranjo.

   `# ip link set veth-noter netns ns-testni` (premakne napravo v prostor `ns-testni`)

   `# ip addr add 10.0.0.1/24 dev veth-zunaj` (nastavi naslov zunanje naprave)

   `# ip netns exec ns-testni ip addr add 10.0.0.2/24 dev veth-noter` (nastavi

[10] Program `ip` je v sodobnih operacijskih sistemih GNU/Linux zamenjava za program `ifconfig`, ki se je pred njim uporabljal za nastavljanje omrežnih naprav.

naslov notranje naprave)
`# ip netns exec ns-testni ip link set veth-noter up` (vključi notranjo napravo)

4. Nato pa samo še poskusimo, na primer s programom `ping`, če povezava deluje. Prvi poskus je povezava od zunaj do naprave v imenskem prostoru, ki ima naslov IP `10.0.0.2`:

```
$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.068 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.077 ms
...
```

   Povezava deluje, saj dobimo paketke nazaj. Poskusimo `ping` iz novega imenskega prostora navzven:

```
# ip netns exec ns-testni ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.066 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.072 ms
...
```

   Tudi tukaj smo prejeli paketke, tako da pošiljanje iz notranjega imenskega prostora deluje.

Opisana konfiguracija ni dovolj, da bi se programi znotraj novega imenskega prostora lahko povezali na internet, lahko pa na ta način z njimi komunicirajo lokalni programi. Če bi jih želeli povezati na internet potrebujemo bolj zapleteno konfiguracijo, na primer t. i. mostove (angl. *bridge*) [48] ali posredovanje IP naslova in uporabo tehnologije NAT (angl. ***N**etwork **A**ddress **T**ranslation*) [28], kar pa ni v obsegu tega dela.

Uporaba imenskih prostorov omrežij je še ena sestavina lahke virtualizacije, posameznemu neprivilegiranemu procesu pa lahko popolnoma onemogočimo dostop do interneta tako, da ga postavimo v nov imenski prostor omrežij, saj ne bo imel nastavljenih omrežnih naprav.

2.3.4 Priklopi v datotečnem sistemu

Imenski prostori priklopov (angl. *mount namespaces*) omogočajo ločevanje podatkov o priklopih v datotečnem sistemu med procesi iz različnih prostorov. Na primer, dva procesa lahko vidita različne imenike in datoteke pod domačim imenikom uporabnikov na `/home`. Na priklopni točki so lahko vidni podatki ene od pomnilniških naprav, lahko je povezovalni priklop na drugo mesto v drevesu datotečnega sistema, lahko je začasen datotečni sistem tipa `tmpfs`, ki se samodejno izbriše skupaj z imenskim prostorom, ali pa katerakoli druga vrsta priklopa.

Nov imenski prostor priklopov ustvarimo podobno kot ostale, s sistemskima klicema `clone` ali `unshare`, podamo pa jima zastavico `CLONE_NEWNS`. Pripona imena zastavice, `NS`, pomeni kar *namespace*. Razlog za to je zgodovinski, saj so ti imenski prostori najstarejši in so bili implementirani prvi, ob njihovi zasnovi pa ni bilo načrtovano, da bodo obstajali še drugi.

Seznam priklopov, ki so del imenskega prostora, lahko preberemo iz treh datotek v imeniku `/proc`. Vsebina vsake datoteke je sestavljena iz vrstic, kjer vsaka predstavlja en priklop v drevesu datotečnega sistema.

1. Datoteka `/proc/mounts`, ki vsebuje seznam priklopov v imenskem prostoru procesa, ki datoteko prebere. Vsebina je oblikovana v formatu datoteke `fstab`, ki sicer vsebuje zače-

tno stanje priklopnih točk ob zagonu sistema [27].

2. Datoteka `/proc/PID/mounts`, ki ima enak format kot prejšnja, ampak prikaže podatke poljubnega procesa. Prejšnja datoteka je simbolična povezava na `/proc/self/mounts`.
3. Datoteka `/proc/PID/mountstats`, ki za prikaz podatkov uporablja svoj, enostavnejši format. Za druge procese jo lahko prebere samo proces s pravicami super uporabnika. Naslednja vrstica je primer iz te datoteke:

   ```
   device /dev/sda4 mounted on /home with fstype ext4
   ```

   Vrstica pomeni, da je naprava `/dev/sda4` priklopljena na priklopno točko `/home` v drevesu datotečnega sistema in da je njen datotečni sistem vrste `ext4`. Sama naprava `sda4` pomeni četrto particijo (zato `4`) prve pomnilniške naprave na sistemu (zato `a` v `sda`).

Ko je ustvarjen nov imenski prostor priklopov se seznam priklopov skopira od njegovega prednika. Spremembe priklopov v imenskem prostoru se lahko posredujejo v ostale imenske prostore priklopov, odvisno od načina deljenja, ki se nastavi za vsak priklop posebej. Način deljenja vpliva tudi na povezane priklope iz istega imenskega prostora. Različni načini so:

1. *skupni* priklop (angl. *shared mount*). V tem načinu se vse spremembe v poddrevesu priklopa kopirajo v povezane priklope iz drugih imenskih prostorov. Učinek je dvosmeren, tako da lahko spremenimo poddrevo priklopa iz kateregakoli imenskega prostora in bo sprememba vidna v vseh ostalih prostorih. Na ta način ostane vsebina priklopov med seboj vedno enaka.
2. *Podrejeni* priklop (angl. *slave mount*) je enak skupnemu, razen tega, da se spremembe iz ostalih povezanih priklopov posredujejo samo vanj, on pa jih ne posreduje ostalim.
3. *Zasebni* priklop (angl. *private mount*) je takšen, ki sprememb niti ne posreduje niti ne sprejema.
4. *Nepovezljivi* priklop (angl. *unbindable mount*) je enak zasebnemu, s to dodatno omejitvijo, da z njim ne moremo delati novih povezovalnih priklopov.

Privzeti način deljenja priklopov je zasebni, in je enak tistemu, ki je bil implementiran pred obstojem imenskih prostorov priklopov. Mehanizem deljenja priklopov lahko uporabljamo tudi brez imenskih prostorov, saj se način deljenja med povezanimi priklopi upošteva tudi znotraj istega imenskega prostora.

Način deljenja priklopov lahko nastavimo s pomožnim programom `mount` [35], ki ima za to zastavice `--make-shared`, `--make-slave`, `--make-private` in `--make-unbindable` ter njihove rekurzivne različice (`rshared`, `rslave`, `rprivate` in `runbindable`), ki ne učinkujejo samo na dan priklop, temveč tudi na vse priklope v njegovem poddrevesu. Za nastavljanje priklopov lahko uporabimo tudi sistemski klic `mount`, ta pa za to sprejema zastavice `MS_SHARED`, `MS_SLAVE`, `MS_PRIVATE` in `MS_UNBINDABLE`, ki pa v njegovem priročniku še niso dokumentirane [34].

Če uporabljamo imenske prostore priklopov za lahko virtualizacijo moramo biti previdni pri uporabi skupnega načina deljenja priklopov. Če je priklop korena datotečnega sistema nastavljen kot skupni, bodo programi znotraj virtualiziranega okolja lahko vplivali na priklope zunanjih programov, zato je takoj po zagonu virtualiziranega okolja potrebno deljenje priklopnih točk znotraj okolja nastaviti na podrejeno ali zasebno.

Ker je privzeti način deljenja zasebni to običajno ne bi predstavljalo težav, vendar program `systemd`, ki v mnogih distribucijah GNU/Linux zamenja `init`, ob zagonu priklopno točko korena nastavi na skupno [44]. To stori zato, da lahko virtualizirana okolja privzeto sprejema-

jo nove priklopne točke iz korena, na primer pogon za zgoščenke ali pa pomnilniške naprave USB.

Kot ostali imenski prostori je tudi imenski prostor priklopov pomembna sestavina sistemov za lahko virtualizacijo.

2.3.5 Imena UTS

Najenostavnejši imenski prostori so imenski prostori UTS. Njihovo ime je zgodovinsko in predstavlja ime ene izmed implementacij operacijskega sistema Unix za velike računalnike IBM (***U**niversal **T**imesharing **S**ystem*). Med procesi iz različnih imenskih prostorov skrijejo samo dva podatka: ime gostitelja (angl. *hostname*) in domensko ime NIS (angl. ***N**etwork **I**nformation **S**ervice domain name*).

Podatka se uporabljata pri sinhronizaciji podatkov uporabnikov in podatkov strežnikov v bolj zapletenih poslovnih omrežjih. Sodobni sistemi namesto tehnologije NIS uporabljajo LDAP (***L**ightweight **D**irectory **A**ccess **P**rotocol*), saj nudi porazdeljeno arhitekturo in višjo varnost.

V operacijskih sistemih GNU/Linux lahko vir podatkov uporabnikov za overjanje nastavimo s funkcionalnostjo NSS (*Name Service Switch*). Nastavitve virov se nahajajo v datoteki `/etc/nsswitch.conf`, kjer za posamezno vrsto podatkov, kot so imena uporabnikov (navadno v `/etc/passwd`), gesla uporabnikov (navadno v `/etc/shadow`), znane gostitelje (navadno v `/etc/hosts`) in podobne nastavimo enega ali več virov. Privzeti vir so že omenjene nastavitvene datoteke, lahko pa izberemo tudi LDAP, DNS (***D**omain **N**ame **S**ystem*), NIS, NIS+ (nadgradnja NIS), WINS (***W**indows **I**nternet **N**ame **S**ervice*) in druge, ki jih je možno namestiti modularno kot dodatke.

2.3.6 Uporabniki

Imenski prostori uporabnikov (angl. *user namespaces*) med procesi iz različnih prostorov ločijo podatke za preverjanje dovoljenj uporabnikov. Sem sodijo identifikatorji uporabnikov (vrednosti `uid`), identifikatorji skupin (vrednosti `gid`), ključi v jedru (sistemski klic `keyctl` [33]) in t. i. zmožnosti uporabnikov (angl. *capabilities*, zastavice, ki privilegije super uporabnika razdelijo na manjše enote [31]).

Procesova `uid` in `gid` sta lahko zunaj in znotraj imenskega prostora uporabnikov različna. Znotraj imenskega prostora lahko ima proces privilegiran (super uporabnikov) dostop do virov, medtem ko ga zunaj nima. V tem primeru je njegov `uid` v prostoru 0, zunaj prostora pa različen od 0.

Vsak proces sodi v natanko en imenski prostor uporabnikov. Če ob nastanku ne ustvari novega prostora, postane član prostora svojega prednika. Nov prostor se ustvari s sistemskima klicema `clone` ali `unshare` in zastavico `CLONE_NEWUSER`, premik v prostor drugega procesa pa je možen s sistemskim klicem `setns`.

Ob vstopu procesa v drug imenski prostor uporabnikov, ki je lahko nov ali pa že obstoječ, postane njegova množica zmožnosti polna. To pomeni, da lahko neomejeno upravlja z vsemi sistemskimi viri znotraj imenskega prostora. Če ima proces polno množico zmožnosti je to skoraj enako, kot bi imel `uid` 0 (bi bil super uporabnik), vendar ima v našem primeru `uid` 65534, ki je neobstoječa številka uporabnika, oziroma vrednost shranjena v `proc/sys/kernel/overflowuid`. Njegov začetni `gid` bo vrednost iz `/proc/sys/kernel/overflowgid`, privzeto tudi 65534. To vrednost jedro uporabi takrat, ko proces znotraj imenskega prostora nima definiranih preslikav `uid` in `gid`.

S preslikavami identifikatorjev definiramo, katere vrednosti `pid` in `gid` ima proces znotraj in zunaj imenskega prostora uporabnikov. Preslikave se nahajajo v datotekah `/proc/PID/uid_map` in `/proc/PID/gid_map`. V novem imenskem prostoru sta datoteki prazni, s pisanjem v njih pa lahko preslikave nastavimo. Vsebina datotek je sestavljena iz posameznih vrstic s tremi podatki: `uid` (oziroma `gid`) znotraj imenskega prostora, `uid` (oziroma `gid`) zunaj imenskega prostora in število preslikanih vrednosti. Število preslikanih vrednosti omogoča, da hkrati preslikamo več kot en par `uid` oziroma `gid`. Od navedenega identifikatorja naprej se preslika toliko identifikatorjev, kot določa število. Razponi identifikatorjev se ne smejo prekrivati, trenutno največje dovoljeno število vrstic pa je pet (vsaj do različice jedra 3.17.4).

Iz varnostnih razlogov je pisanje v datoteke preslikav dovoljeno samo enkrat v življenjski dobi imenskega prostora uporabnikov. Vanjo smejo pisati samo procesi iz prednikovega imenskega prostora in sam proces znotraj imenskega prostora, ki pa lahko ustvari preslikavo samo na `uid` svojega neposrednega prednika. Identifikatorji, ki jih proces pri ustvarjanju preslikav uporabi, so relativni na njegov prostor. To pomeni, da za identifikatorje znotraj prostora uporablja vrednosti, ki zares sodijo v njegov imenski prostor, medtem ko za identifikatorje zunaj prostora uporablja vrednosti, ki imajo smisel v imenskem prostoru prednika. Ta pravila omogočajo, da si lahko proces znotraj novega imenskega prostora uporabnikov sam nastavi preslikave identifikatorjev na uporabno vrednost, na primer tako, da za `uid` znotraj prostora nastavi številko 0 (in postane znotraj prostora super uporabnik), za `uid` zunaj prostora pa `uid` procesa, ki je nov prostor ustvaril.

Imenski prostori uporabnikov so nadrejeni ostalim imenskim prostorom. Vsak imenski prostor ima referenco na svoj imenski prostor uporabnikov. Ko proces poskuša dostopati do nekega vira, se najprej preveri, ali je imenski prostor vira del imenskega prostora uporabnikov tega procesa. Če ni, je dostop zavrnjen. V nasprotnem primeru se upoštevajo običajni načini za preverjanje dostopa. Tako z imenskimi prostori uporabnikov omejimo dostop do vseh virov, ki sodijo pod okvir imenskih prostorov. Če proces ne ustvari novih imenskih prostorov za ostale vire, virov sploh ne more uporabljati. Procesi v nadrejenem imenskem prostoru uporabnikov imajo poln dostop do vseh virov iz podrejenih prostorov.

Razen te posebnosti se delovanje imenskih prostorov priklopov (razdelek 2.3.4) dodatno spremeni. Vse priklopne točke, ki so nastavljene kot skupne, postanejo znotraj imenskega prostora uporabnikov podrejene. To onemogoči prenos sprememb po imenskih prostorih navzgor in je varnostno nujno, saj bi sicer neprivilegiran uporabnik lahko ustvaril nov imenski prostor uporabnikov in spreminjal skupne priklope izhodiščnega imenskega prostora sistema.

V splošnem je smiselno, da uporabo teh imenskih prostorov vedno spremljajo še ostali imenski prostori, katerih vire imamo namen uporabljati. V trenutni implementaciji lahko v nasprotnem primeru naletimo na težave s kompatibilnostjo. Na primer, nekateri mehanizmi za medprocesno komunikacijo uporabljajo `uid` za overjanje uporabnika, ga pa ne preslikajo glede na imenski prostor, zato je uporaba imenskih prostorov za medprocesno komunikacijo skoraj obvezna. Podobno velja za navidezen datotečni sistem, ki ob dostopu do imenikov in datotek preverja nepreslikan `uid`, zato moramo zaradi varnosti uporabiti tudi imenske prostore priklopov.

S pomočjo imenskih prostorov uporabnikov lahko implementiramo lahko virtualizacijo brez potrebe po pravicah super uporabnika, saj ima proces znotraj novega prostora vse zmožnosti. Posledično so se pri začetnih implementacijah funkcionalnosti pojavile ranljivosti, ki so neprivilegiranim procesom omogočale pridobitev pravic super uporabnika na gostitelju [1, 12,

14, 40]. Zaradi tega večina distribucij GNU/Linux imenskih prostorov uporabnikov privzeto ne vključi v jedro, saj njihovemu delovanju še ne zaupajo.

Prav tako je bil za njihovo implementacijo potreben velik poseg v delovanje jedra. Ena večjih sprememb je zagotovo uvedba internih identifikatorjev `kuid` in `kgid`, ki predstavljajo interne različice `uid` in `gid` za uporabo v jedru, ki ju ni potrebno preslikovati med prostori. Zaradi njihove uvedbe morajo vsi podsistemi, ki za dovoljenja uporabljajo identifikacijo uporabnikov, implementirati podporo za njih. Takšni so na primer vsi datotečni sistemi, saj uporabljajo podatek o lastništvu datotek za preverjanje dovoljenj.

Imenski prostori uporabnikov so zmogljivost jedra, ki se skupaj z ostalimi imenskimi prostori najbolj približuje elegantni rešitvi težave diplomskega dela. Razlog je predvsem v tem, da lahko z njimi pripravimo čisto, ponovljivo in varno okolje za izvajanje drugih programov, pri tem pa ne potrebujemo pravic super uporabnika, kar bistveno izboljša splošno uporabnost in varnost rešitve. Vendar pa je uporabnost funkcionalnosti omejena, saj v večini distribucij GNU/Linux privzeto ni na voljo, tako da je potrebno ročno nastavljanje, prevajanje in namestitev jedra. Prav tako ni zagotovljeno, da je bila večina varnostnih ranljivosti že odkritih.

2.4 Nadzorne skupine

Nadzorne skupine (angl. *control groups*) so mehanizem, ki množice opravil (množice procesov in niti) združujejo v hierarhično organizirane skupine. Te skupine lahko podsistemi nadzornih skupin uporabljajo za omejevanje uporabe sistemskih virov opravil, lahko pa z njimi tudi podrobno sledijo njihovemu delovanju. Na primer, izvajanje opravil iz skupine lahko omejijo na določena jedra centralne procesne enote, omejijo jim lahko uporabo delovnega pomnilnika ali nastavijo prednost pri razvrščanju dostopa do vhodno izhodnih enot. Omogočajo tudi vpogled v izvajanje opravil, kot je poraba časa na centralni procesni enoti ali statistiko o porabi delovnega pomnilnika. Nadzorne skupine najpogosteje srečamo pod krajšim imenom »cgroups«.

Na sistemu lahko deluje več hierarhij nadzornih skupin hkrati, kjer je vsaka organizirana kot drevo navideznega datotečnega sistema. Vsak imenik v drevesu predstavlja svojo nadzorno skupino, celotna hierarhija pa vedno vsebuje vse procese na sistemu. To pomeni, da lahko z uporabo večih hierarhij procese organiziramo na različne načine, razlikujejo pa se tudi po vključenih podsistemih. Hierarhije običajno najdemo kot podimenike `/sys/fs/cgroup`, vendar ta lokacija ni standardna in se lahko med distribucijami in uporabniki razlikuje. Novo hierarhijo nadzornih skupin lahko ustvarimo s sistemskim klicem `mount` z argumentom za tip datotečnega sistema `cgroup` in dodatnimi argumenti, ki določajo vključene podsisteme. Za skrbništvo nad nadzornimi skupinami ni posebnih sistemskih klicev, ampak jih nastavljamo z dodajanjem in brisanjem imenikov v hierarhiji ter s pisanjem v posebne datoteke, ki se v teh imenikih pojavijo. Za to lahko uporabljamo običajne ukaze in funkcije (`mkdir` za dodajanje, `rmdir` za odstranjevanje, `mv` za preimenovanje, `cat` za branje in tako dalje).

Katere posebne datoteke so na voljo v imenikih nadzornih skupin je odvisno od vključenih podsistemov, nekatere pa so na voljo vedno. Takšna je `cgroup.procs`, ki vsebuje seznam vseh `pid` procesov pod okriljem nadzorne skupine. Proces v skupino postavimo tako, da v datoteko zapišemo njegov `pid`[11]. Podobno obstaja datoteka `tasks`, ki deluje z identifikatorji niti (`tid`) namesto procesov. Iz skupine opravila ne moremo odstraniti, lahko pa ga premaknemo v

[11] V datoteke lahko zapišemo samo en identifikator naenkrat, saj se pisanje vedno opravi s sistemskim klicem `write`, ki v primeru napake ne more vrniti več kot enega statusa.

drugo skupino. Prav tako vsebuje vsak imenik datoteko `notify_on_release`, ki določa, ali bo operacijski sistem sprožil obvestilo, ko se zadnje opravilo v nadzorni skupini zaključi. Če je njena vsebina 0, sporočila ne bo, z 1 pa ga vključimo. Ravnanje s samim sporočilom je določeno z datoteko `release_agent` v korenskem imeniku hierarhije, ki vsebuje pot do programa za odziv ob izpraznitvi nadzorne skupine.

Hierarhija vsebuje poleg nadzornih skupin običajno še podsisteme za nadzor ali vpogled v procese iz skupin, lahko pa jo uporabljamo tudi brez njih. Ker nam mehanizem zagotavlja, da novi procesi in niti, ki jih ustvarijo obstoječi, ohranijo članstvo v enakih nadzornih skupinah, lahko držimo pregled nad procesi, ki postanejo demoni[12]. Ker je mehanizem nadzornih skupin ločen od drevesa procesov, ostane takšen demon v enaki skupini kot prej, čeprav ga je posvojil `init`. To bistveno olajša iskanje procesov, ko želimo ustaviti vse procese nekega začetnega procesa.

Z vključitvijo dodatnih podsistemov dobimo možnost naprednejšega nadzora in pregleda nad procesi. Vsak podsistem je lahko naenkrat vključen v samo eni hierarhiji nadzornih skupin hkrati. Njihovo delovanje nastavljamo z novimi posebnimi datotekami, ki se v imenikih nadzornih skupin pojavijo. Različni podsistemi nadzornih skupin so naslednji.

1. Podsistem `blkio` je nadzornik, ki omogoča nadzor in pregled nad vhodno izhodnimi napravami [15]. Z njim lahko posameznim nadzornim skupinam v hierarhiji nastavimo uteži, s katerimi se opravilom iz skupine določa prednost pri dostopu do naprav. Dostop lahko omejimo tudi z določitvijo zgornje meje prepustnosti podatkov v bajtih na sekundo. Nadzornik prav tako omogoča pregled statistike uporabe vhodno izhodnih naprav, kot je na primer skupen čas dostopa opravil skupine do sedaj ali število prenesenih sektorjev.
2. Sorodna podsistema `cpuset` in `cpuacct` omogočata nadzor in pregled nad uporabo centralnih procesnih enot [18, 17]. Z nadzornikom `cpuset` lahko izvajanje opravil omejimo na točno določena jedra procesorja, njihovo uporabo delovnega pomnilnika pa lahko omejimo na izbrane fizične module. Takšne zmožnosti so najbolj uporabne pri velikih računalnikih, kjer že razdalje med računskimi jedri in pomnilniškimi moduli niso trivialne. Takrat lahko delovanje sistema optimiziramo z omejitvijo programov na en del računalnika. Funkcionalnost pa je uporabna tudi, če ne želimo, da nek računsko zahteven program upočasni vse ostale programe na osebnem računalniku.

   Podsistem `cpuacct` zbira podatke o porabi procesorskega časa opravil v nadzorni skupini. Rezultat njegovega dela je podoben izpisu ukaza `time`, ki ob zaključku programa izpiše realne in sistemske čase izvajanja programa. Razlika je, da `cpuacct` to statistiko nudi na nivoju nadzornih skupin, rezultate pa lahko beremo tudi med izvajanjem programa.
3. Podsistem `devices` opravilom nadzorne skupine omejuje pravice dostopa do posebnih znakovnih in bločnih naprav v navideznem datotečnem sistemu [19]. To so na primer naprave `/dev/null`, `/dev/random`, `/dev/sda2` in podobne. Za vsako posamezno napravo lahko nastavimo pravico do pisanja, branja in ustvarjanja, privzeto pa so vsi dostopi odobreni.
4. Podsistem `freezer` omogoča zamrznitev vseh opravil v hierarhiji nadzornih skupin [20]. Mehanizem deluje podobno kot pošiljanje signala `SIGSTOP` vsem procesom v skupini, ven-

[12] Demoni (angl. *daemon*) so procesi, ki jih namenoma naredimo v sirote, da jih posvoji `init`. Tako lahko nemoteno tečejo v ozadju.

dar se opravi tako, da procesom ni viden. Pri uporabi signalov je za nadaljevanje izvajanja procesu potrebno poslati signal `SIGCONT`, s katerim lahko procesi ravnajo po svoje.

Mehanizem je bil zasnovan zaradi uporabe v gručah računalnikov (angl. *computing cluster*), saj ob zamrznitvi ohrani vse podatke potrebne za nadaljevanje izvajanja programov na drugih fizičnih računalnikih. Z njegovo pomočjo lahko množico tekočih programov zamrznemo in jih prenesemo na drug računalnik, kjer njihovo izvajanje nadaljujemo. Tega posega pa procesi ne vidijo.

5. S podsistemom `hugetlb` lahko nadzorujemo uporabo velikih strani (angl. *huge pages*) pri ostranjevanju pomnilnika [21]. To so pomnilniške strani večje od običajnih[13], za katere lahko programi posebej zaprosijo.
6. Podsistem `memory` opravilom omeji uporabo delovnega pomnilnika [22]. Omejitve nastavimo posebej za sam delovni pomnilnik procesa (sklad in kopica) in za količino izmenjanega pomnilnika[14].

   Če eno od opravil omejitev preseže, se izvede običajen postopek v primeru pomanjkanja pomnilnika, le da učinkuje samo nad procesi znotraj nadzorne skupine, in ne nad celotnim sistemom. Ta postopek, imenovan `oomkill` (angl. *out of memory kill*), po svojih kriterijih izbere najmanj pomembno opravilo in ga zaključi.

   Omejitve lahko nastavimo tudi na porabo delovnega pomnilnika notranjih struktur jedra. To vsebuje podrobnosti kot so podatki nekaterih omrežnih protokolov in pomnilniške strani skladov. Vendar v dokumentaciji piše, da je funkcionalnost namenjena samo razvojni in testni uporabi.
7. Zadnja dva podsistema se uporabljata za omrežja. S prvim, `net_prio`, lahko opravilom iz različnih nadzornih skupin nastavimo uteži za dostop do omrežnih vmesnikov [24], kar je podobno kot nadzornik `blkio` omogoča pri vhodno izhodnih napravah. Z drugim, `net_cls`, pa lahko paketkom iz omrežnega prometa dodelimo značke glede na njihovo članstvo v nadzorni skupini [23]. Ostali omrežni programi lahko te značke vidijo in z njimi sprejemajo odločitve. Na primer, pravila za požarne zidove (angl. *firewall*) lahko s programom `iptables` podrobneje nastavimo glede na članstvo procesa v nadzorni skupini. Podobno lahko storimo s programom `tc`, ki zna značke uporabljati pri nastavljanju prednosti omrežnih paketkov.

Nadzorne skupine so uporabno orodje skrbnikov strežnikov spletnih strani, saj lahko za posamezno stran določijo različne omejitve do sistemskih virov in preprečijo morebitno preobremenitev sistema. V sodobnih distribucijah jih najpogosteje srečamo skupaj s programom `systemd`, ki ob zagonu sistema v imeniku `/sys/fs/cgroup` ustvari po eno hierarhijo za vsak podsistem nadzornih skupin, prav tako pa ustvari dodatno hierarhijo brez vključenih podsistemov, ki jo uporablja za sledenje procesom posameznih sej vseh uporabnikov. Na ta način nadgradi obstoječ način razvrščanja procesov, ki je omejen na štiri nivoje identifikatorjev:

1. `sid` (angl. ***s**ession **id**entifier*) je identifikator tistega procesa, ki predstavlja vodjo seje. Nova seja se ustvari, ko zaženemo nov primerek lupine oziroma s sistemskim klicem `setsid`.

[13] Velikost pomnilniške strani sodobnega procesorja v osebnem računalniku je običajno 4 kilobajte, velike strani pa dosegajo velikosti do 16 gigabajtov.

[14] Izmenjani pomnilnik (angl. *swap*) je pomnilnik, ki se uporabi v primeru pomanjkanja hitrega delovnega pomnilnika. Njegovi podatki se običajno nahajajo globlje v pomnilniški hierarhiji, na večjih obstojnih pomnilniških medijih.

Vsi procesi z istim `sid` si delijo enak terminal ali pa terminala nimajo. Procesi demoni, ki vedno tečejo v ozadju in niso vezani na terminal, običajno ustvarijo svojo lastno sejo.

2. `pgid` (angl. ***p**rocess **g**roup **id**entifier*) je identifikator tistega procesa, ki predstavlja vodjo skupine procesov. Vsaka seja lahko vsebuje več skupin procesov, novo pa proces ustvari s sistemskim klicem `setpgid`. To je koncept, ki ga uporabljajo lupine za združevanje procesov v večje naloge (angl. *jobs*). Na primer, ko ustvarimo cev programov so vsi del ene takšne skupine. V lupini je v ospredju vedno samo ena skupina procesov, in je tista, ki prejema vnos s tipkovnice. Skupine lahko izpišemo z ukazom `jobs`, jih postavimo v ospredje z ukazom `fg` ali pošljemo v ozadje z ukazom `bg`.
3. `pid`, ki ga že poznamo, je tretji nivo identifikatorjev in predstavlja en proces v skupini procesov.
4. `tid`, ki smo ga tudi že srečali, je četrti nivo identifikatorjev in predstavlja eno nit enega procesa.

Z uporabo hierarhije nadzornih skupin za organiziranje procesov se `systemd` izogne omejitvam tega štiri nivojskega načina razvrščanja. Zanj je največja omejitev to, da lahko poljuben proces vedno ustvari svojo sejo s sistemskim klicem `setsid` in s tem začne svojo lastno razvrstitev, ki več ni vezana na druge seje. Na svojo razvrstitev v hierarhiji nadzornih skupin sami neprivilegirani procesi ne morejo vplivati, ta pa ostane stalna ne glede na njihova članstva v sejah ali skupinah procesov. Prav tako so novi procesi samodejno člani enakih nadzornih skupin kot njihovi starši, tako da `systemd` na njih ne rabi biti posebno pozoren.

Nadzorne skupine v veliki meri uporablja z drugačnim namenom tudi projekt LXC, ki je podrobno opisan v razdelku 2.10.

Same nadzorne skupine so prisotne v jedru že relativno dolgo (od različice 2.6.24 naprej), vendar je bil takrat na voljo samo podsistem za nadzor centralnih procesnih enot. Njihovo širše sprejetje se je začelo komaj s sprejemom programa `systemd` kot nadomestilo programa `init`, in s priljubljenostjo programa LXC za lahko virtualizacijo. Od takrat so razvijalci prišli do spoznanja, da je trenutni model hierarhij preveč splošen in zato prezahteven za elegantno in učinkovito implementacijo. Zato obstaja novejša oblika hierarhij, imenovana *poenotena hierarhija* (angl. *unified hierarchy*) [16]. Ta je v sodobnih jedrih privzeto sicer že na voljo (od različice 3.16 naprej), ampak le kot dodatna razvojna zastavica sistemskemu klicu `mount`, saj bo lahko še deležna sprememb.

Poenotena hierarhija se od sedanjega stanja razlikuje tako, da več ne moremo imeti več kot ene hierarhije nadzornih skupin, kot se zdaj nahajajo v imeniku `/sys/fs/cgroup`, temveč samo eno poenoteno hierarhijo. S tem se zmanjša prilagodljivost nadzornih skupin, pridobimo pa enostavnejšo implementacijo in hitrejše delovanje mehanizmov. Razlog je tudi, da so prednosti trenutne prilagodljivosti hierarhij vprašljive. Posamezen podsistem je lahko del samo ene hierarhije, zato njihova nastavitev običajno vsebuje po eno hierarhijo za vsak podsistem, kar pa povzroči potrebo po zrcaljenju organizacije procesov v vsaki hierarhiji, če jih želimo nadzirati z različnimi podsistemi hkrati [3]. Spremeni se tudi to, da lahko v poenoteni hierarhiji na nivoju vsake nadzorne skupine določimo, kateri podsistemi so za njo vključeni, kar povrne nekaj izgubljene prilagodljivosti pri organizaciji procesov.

Nadzorne skupine so uporabna sestavina za lahko virtualizacijo in uporaben mehanizem v okviru diplomskega dela. Skupaj z uporabo imenskih prostorov procesi ne vidijo kakšnega nadzora so deležni v okviru nadzornih skupin, iz omejenega okolja pa ne morejo izstopiti.

2.5 Način računanja seccomp

seccomp (angl. ***secure comp**uting mode* ali varni način računanja) je mehanizem v jedru Linux, ki procesu onemogoči uporabo določenih sistemskih klicev. Če proces pokliče prepovedan sistemski klic, ga jedro prekine. Ker brez uporabe sistemskih klicev proces ne more komunicirati z zunanjim okoljem, kar vsebuje ostale procese, datotečni sistem, omrežje in podobno, mu takšna prepoved onemogoči vsako možnost škodovanja sistemu. Vendar brez komunikacije z okoljem tudi svojih rezultatov ne more sporočiti in s tem narediti česa uporabnega, zato moramo nekatere sistemske klice dovoliti.
Mehanizem seccomp se vključi s sistemskim klicem prctl (angl. ***pr**ocess **con**trol*), ki omogoča nizkonivojske operacije nad procesi [37]. Za seccomp mu podamo argument PR_SET_SECCOMP, učinkuje pa na proces, ki prctl pokliče. Dovoljene sistemske klice določimo z načinom delovanja. Ta je lahko SECCOMP_MODE_STRICT, ali strogi način, ki dovoli nekaj vnaprej določenih sistemskih klicev in v primeru ostalih procesu pošlje signal KILL, ali SECCOMP_MODE_FILTER oziroma način s filtriranjem, ki omogoča nastavitev različnih odzivov na posamezne sistemske klice. V strogem načinu so dovoljeni naslednji sistemski klici:

1. read, ki procesu omogoča, da bere datoteke. Ker je datoteke pred branjem potrebno odpreti (na primer s sistemskim klicem open, ki v strogem načinu ni dovoljen) s tem sistemskim klicem ni mogoče narediti velike škode sistemu. To omogoča zunanjemu nadzornemu procesu komunikacijo z omejenim procesom preko že odprtih datotek. Obstaja pa enaka nevarnost kot pri chroot (razdelek 2.2), če nadzorni proces ne zapre datotečnih deskriptorjev, ki jih omejen proces ne sme brati.
2. write, ki procesu omogoča, da piše v datoteke. Varnostne skrbi so enake kot pri sistemskem klicu read.
3. sigreturn, ki ga proces potrebuje zaradi notranjega delovanja jedra Linux. Uporablja se za implementacijo signalov. Ko proces prejme signal, mu jedro v okvir sklada rutine (angl. *stack frame*) za ravnanje s signalom vstavi klic sigreturn. Ob zaključku izvajanja funkcije se pokliče, da za sabo počisti okvir s podatki signala.
4. _exit, ki ga mora izvesti vsak proces, ko zaključi z izvajanjem. Z njim sporoči izhodni status (število v velikosti enega bajta) procesu, ki ga je ustvaril. Status je običajno 0 za uspešen zaključek in nekaj drugega v primeru napake. Programer mora biti tukaj previden, saj funkcija exit iz knjižnice glibc od različice 2.3 naprej ne pokliče sistemskega klica _exit ampak sistemski klic exit_group, ki ni dovoljen. Enaka funkcije se v programskem jeziku C pokliče ob vrnitvi iz funkcije main.

Pred jedrom Linux različice 3.5 je bil na voljo samo način delovanja SECCOMP_MODE_STRICT. Ker v tem načinu ne moremo izvesti sistemskega klica execve, ki omogoča zamenjavo procesa z drugim programom, lahko tako omejimo samo izvajanje naših lastnih programov. Kljub tej omejitvi je ta način še vedno uporaben. Če je naš program tolmač tolmačenega programskega jezika, lahko še vedno omejimo obseg programa, saj izvajanje tolmačene kode poteka znotraj našega programa. Strogi način lahko uporabimo tudi preventivno, da omejimo škodo, ki jo lahko povzročijo hrošči v našem lastnem programu.

Z uporabo načina delovanja SECCOMP_MODE_FILTER lahko bolj natančno določimo, kako se bo jedro odzvalo na nedovoljene sistemske klice. Izbiramo lahko med nasilno prekinitvijo procesa (enako signalu KILL), navidezno neuspešno izvedbo klica z nastavitvijo spremenljivke errno, ustavitvijo procesa za uporabo s sistemskim klicem ptrace (razdelek 2.6) in prekinitvijo

procesa s signalom `SIGSYS`. Posamezen sistemski klic pa lahko tudi dovolimo. Prav tako nismo omejeni na štiri sistemske klice iz strogega načina, temveč lahko odziv nastavimo za vsak sistemski klic posebej. Filter izrazimo kot program v jeziku Berkeley Packet Filter (BPF), ki je preprost jezik za opis omejitev. Pred obstojem `seccomp`ovega načina s filtriranjem se je uporabljal samo za filtriranje omrežnih paketkov, kasneje pa je bil razširjen še v ta del jedra. Obstajajo tudi načrti za nadaljnje širitve [2].

Program BPF omogoča določitev sistemskih klicev, ki jih želimo omejiti, in nabor dovoljenih argumentov zanje. Argumente lahko primerjamo samo na nivoju posameznih registrov sistemskega klica, zato ne moremo primerjati podatkov, na katere kažejo kazalci. Med te podatke sodijo tudi običajni nizi znakov. Zaradi tega ne moremo preverjati podatkov, ki jih program izpisuje v konzolo ali piše v datoteke, prav tako pa ne moremo preverjati argumentov, ki predstavljajo poti v datotečnem sistemu. Razlog za to omejitev je varnost. Zlonameren program bi lahko z uporabo vzporedne niti argument spremenil v vmesnem času, ko bi ga `seccomp` že preveril in odobril, ampak še ne izvedel. Napada ne bi bilo enostavno izvesti, saj je osnovan na tveganem stanju (angl. *race condition*), ne smemo pa dovoliti te možnosti. To pomanjkljivost bi bilo mogoče odpraviti, vendar bi za to potrebovali nadgradnjo jezika BPF ali popolnoma drug jezik, kar bi zahtevalo veliko truda s strani razvijalcev jedra. Iz tega razloga so raje priredili obstoječo in preizkušeno tehnologijo. Dodatna omejitev jezika BPF je to, da med preverjanjem posameznih klicev ne hrani nobenega stanja. Zato je nemogoče implementirati pogoj, ki na primer omeji število odprtih datotek, količino porabljenega pomnilnika ali velikost zapisanih datotek na njem.

Ker se med strojnimi arhitekturami razlikuje, v katerem registru je številčni identifikator sistemskega klica in v katerih registrih so njegovi argumenti, je program BPF potrebno implementirati za vsako arhitekturo posebej. Pri tem nam pomaga knjižnica `libseccomp`, ki postopek abstrahira v razumljiv vmesnik neodvisen od arhitekture.

Brskalnik Chromium uporablja `seccomp` za omejitev vtičnika Adobe Flash Player [6], z njim pa omejuje tudi vse podprocese za izris spletnih strani [4, 49]. Čeprav so te podprocese napisali sami programerji Chromiuma, lahko vsebujejo varnostne ranljivosti, zato jih je potrebno dodatno omejiti. Podprocesi za izris strani so zapleteni programi, ki brez dodeljevanja pomnilnika in izrisa na ekran ne morejo delovati. `seccomp` filter, ki vse sistemske klice potrebne za te funkcije dovoli, ne omogoča visoke varnosti, saj ne more preverjati bolj zapletenih argumentov. Zaradi tega so v brskalniku Chromium zasnovali posebno arhitekturo [50], ki pa privzeto ni vključena, saj njena implementacija še ni dokončana.

Ideja nove Chromiumove varnostne arhitekture je sledeča. Uvede se dodaten preverjevalni podprogram z nalogo, da preverja sistemske klice in njihove argumente, ki jih želijo podprocesi za izrisovanje strani izvesti. Vsak podproces je po novem sestavljen iz dveh niti. Prva je nadzorna, ki vsebuje minimalno funkcionalnost za posredovanje želenih sistemskih klicev preverjevalnemu procesu in ki jih z odobritvijo izvede, druga pa vsebuje sam program za izris, ki mu ne zaupamo. Druga nit, ki ji ne zaupamo, sistemskih klicev ne izvaja, ampak za to zaprosi nadzorno nit. Da je lahko to varno, nadzorna nit ne sme uporabljati pomnilnika s spremenljivo vsebino, kar vključuje kopico in celo sklad. V nasprotnem primeru bi jih lahko druga nit spremenila. Zato mora biti nadzorna nit napisana v skrbno preverjenem zbirnem jeziku in izvajati samo posredovanje med nezaupano nitjo in nadzornim procesom. Nadzorni proces obstaja samo zato, ker je takšna preverjanja prezapleteno napisati v zbirnem jeziku.

Arhitektura zahteva, da nezaupane niti ne izvajajo sistemskih klicev neposredno, ampak

za to prosijo nadzorno nit. Ker so vsi sistemski klici abstrahirani v funkcije standardnih knjižnic je to težava, saj bi morali vse funkcije ponovno implementirati. To bi zahtevalo veliko dodatnega dela od programerjev, zato so se odločili za drugo rešitev. Preden se zažene nit s programom za izris, kodo programa z uporabo razgrajevalnika (angl. *disassembler*) na nivoju strojne kode spremenimo tako, da zamenjamo vse sistemske klice s prošnjami nadzorni niti. Ker pretvorbe iz strojne kode v zbirni jezik ni mogoče vedno izvesti pravilno, se lahko zgodi, da se nekateri izmed klicev ne zamenjajo. To je težava, vendar ne vpliva na varnost, saj nezaupana nit teče pod načinom `seccomp`, ki bo nit prekinil ob prvem sistemskem klicu.

Vse to deluje tudi brez `seccomp`ovega naprednejšega načina s filtri, saj lahko komunikacijo med nadzorno in nezaupano nitjo implementiramo s štirimi osnovnimi sistemskimi klici iz strogega načina. Težava se pojavi pri hitrosti, saj komunikacija zahteva čas. Zato Chromium kljub temu uporablja `seccomp`ov način s filtri, da kjer je to mogoče, sistemske klice preverja preko hitrejšega mehanizma v jedru.

Kot že rečeno je ta arhitektura v trenutnih različicah Chromiuma samo testne narave in se ne uporablja v produkciji. Vseeno pa uporablja `seccomp` v načinu s filtri za prepoved sistemskih klicev, ki jih nobena nit načeloma ne potrebuje, saj tako zmanjša možnost napada na samo jedro. Poleg `seccomp`a Chromium uporablja tudi imenski prostor z novim `pid` (razdelek 2.3.1) in `chroot` (razdelek 2.2).

`seccomp` uporablja tudi brskalnik Mozilla Firefox [43] z istim razlogom kot Chromium, za preventivno manjšanje varnostnih ranljivosti v lastni kodi.

Za rešitev težave diplomskega dela sam `seccomp` ni dovolj, saj z njim ni mogoče implementirati vseh omejitev, ki jih rešitev zahteva. Samo z njim ne moremo omejiti dostopa do datotek, ki jih program ne sme videti, ali omejiti sistemskih virov kot sta čas in pomnilnik. Je pa kljub temu uporaben kot del večje rešitve.

2.6 Sistemski klic `ptrace`

Sistemski klic `ptrace` (angl. *process trace* oziroma sledenje procesu) nudi mehanizem, s katerim lahko proces (*sledilec*, angl. *tracer*) spremlja in nadzoruje delovanje drugega procesa (*zasledovani proces*, angl. *tracee*). Sledi mu s preučevanjem in spreminjanjem njegovih registrov in vsebine njegovega delovnega pomnilnika. Prvenstveno je bil namen sistemskega klica implementacija razhroščevalnika, kasneje pa se je njegova uporaba razširila v analizo sistemskih klicev tekočih programov in v simuliranje privilegiranega okolja neprivilegiranim procesom.

Njegov izvor je razviden iz prvih vrstic datoteke `sys/user.h`, ki jo moramo vključiti v svoj program če želimo `ptrace` uporabljati s programskim jezikom C:

```
/* The whole purpose of this file is for GDB and GDB only.  Don't read
   too much into it.  Don't use it for anything other than GDB unless
   you know what you are doing.  */
```

Komentar pomeni, da datoteka obstaja izključno in samo zaradi programa GDB (GNU Debugger), razhroščevalnika operacijskega sistema GNU, in da je ni primerno uporabljati za druge namene. Razen, če vemo točno, kaj počnemo.

S sistemskim klicem `ptrace` začnemo procesu slediti na tri načine. Prvi je koordiniran, kjer na novo ustvarjen proces pokliče `ptrace` z argumentom `PTRACE_TRACEME` in s tem njegovemu predniku omogoči, da mu začne slediti. Takšen proces bo ustavljen dokler mu sledilec ne dovoli nadaljevanja izvajanja. Drugi način je z argumentom `PTRACE_ATTACH`, ki sproži sledenje poljubnega obstoječega procesa in mu pošlje signal `SIGSTOP`, da ga ustavi in s tem omogoči

sledilcu nadzor nad nadaljevanjem. Tretji in najnovejši način je argument `PTRACE_SEIZE`, ki enako kot prejšnji deluje na poljubnem obstoječem procesu, vendar ga ne ustavi takoj. Z njim lahko počakamo, da proces prejme prvi signal, saj takrat dobi nadzor sledilec, lahko pa ponovno pokličemo `ptrace` z argumentom `PTRACE_INTERRUPT` da ga zaustavimo takoj. V vsakem primeru je na zaustavitev procesa potrebno počakati s sistemskim klicem `waitpid`, saj potrdi ustavitev procesa in javi razlog njegove ustavitve.

V ustavljenem stanju lahko sledilec brska po registrih procesa in spreminja njihove vrednosti, prav tako pa lahko vidi in spreminja vsebino pomnilnika. To omogočajo številni argumenti klicu `ptrace`, kot so na primer `PTRACE_PEEKDATA` za prebiranje pomnilnika, `PTRACE_POKEDATA` za pisanje v pomnilnik, `PTRACE_GETREGS` za pridobitev vrednosti vseh registrov in `PTRACE_SETREGS` za spreminjanje njihovih vrednosti. Zgoraj omenjeno datoteko `sys/user.h` potrebujemo zaradi funkcij, ki berejo ali pišejo v registre, saj vsebuje definicije njihovih struktur in konstant.

Po pregledih in spremembah mora sledilec sprožiti nadaljevanje izvajanja procesa, to pa lahko stori na tri načine.

1. `PTRACE_CONT` nadaljuje izvajanje procesa do prvega prejetega signala. Ko proces signal prejme se lahko nanj najprej odzove sledilec. Pogleda lahko, kateri signal je bil poslan in se glede na to odloči, ali bo signal posredoval procesu ali mu ga bo zamolčal.
2. `PTRACE_SYSCALL` tako kot `PTRACE_CONT` ustavi proces ob prejetem signalu, ustavi pa ga prav tako tik pred izvedbo vsakega sistemskega klica. V tem trenutku lahko sledilec z uporabo branja registrov in pomnilnika ugotovi, kateri sistemski klic je proces želel izvesti in s kakšnimi argumenti. Če sledilec podatke spremeni, se bo izvedel spremenjen sistemski klic. Po nadaljevanju izvajanja bo proces ponovno ustavljen tik pred zaključkom izvedbe sistemskega klica. Takrat lahko sledilec pregleda rezultat sistemskega klica (njegov uspeh, količino zapisanih podatkov in podobno), možnost pa ima rezultat spremeniti in tako »lagati« procesu.
3. `PTRACE_SINGLESTEP` dovoli izvedbo samo enega strojnega ukaza, nato pa proces ustavi. S tem načinom lahko implementiramo običajno zmožnost sestopanja skozi program, ki ga omogočajo razhroščevalniki. Prav tako pa bo proces kot pri ostalih dveh načinih zaustavljen v primeru signala.

2.6.1 Primeri uporabe

Za razhroščevalnike je `ptrace` orodje, s katerim opravijo najpomembnejši del svojega delovanja. Z njim lahko tekoče programe poljubno ustavijo in pregledajo, njihove podatke pa prikažejo uporabnikom. Spreminjajo lahko njihove registre in vrednosti spremenljivk v delovnem pomnilniku. Vstavijo lahko tudi prekinitvene točke (angl. *breakpoint*), kar naredijo tako, da na določenih mestih kodo programa zamenjajo s svojo. Brez uporabe sistemskega klica `ptrace` je sicer razhroščevalnik mogoče napisati, vendar takšni niso v širši uporabi. Takšen pristop temelji na uporabi datoteke `/proc/PID/mem`, ki privilegiranim procesom dovoli dostop do pomnilnika vsakega procesa [47].

Program `strace` je zanimivo a manj znano orodje, ki uporablja `ptrace`. Njegovo ime pomeni *trace system calls* ali sledenje sistemskih klicev. Program izpiše vse sistemske klice, ki jih poljuben program izvaja, kar je uporabno za diagnostične, razhroščevalne in poučne namene. Na primer, če ga uporabimo za ukaz `/bin/echo test`, ki na ekran izpiše besedilo

`test`, dobimo:

```
execve("/bin/echo", ["/bin/echo", "test"], [/* 36 vars */]) = 0
[...]
write(1, "test\n", 5)                   = 5
[...]
exit_group(0)                           = ?
+++ exited with 0 +++
```

Prva vrstica predstavlja sistemski klic `execve` in njegove argumente. Program `echo` tega klica ne uporablja, njegov izpis je posledica delovanja programa `strace`, saj za začetek sledenja uporabi prvi, koordiniran način. To pomeni, da najprej ustvari nov proces, ki `ptrace` pokliče z argumentom `PTRACE_TRACEME`, nato pa ta isti proces pokliče `execve`, da svojo kodo zamenja s kodo programa, ki ga želimo izvesti. Ker je takoj po uporabi `PTRACE_TRACEME` proces ustavljen, poteka izvedba klica `execve` že v okviru sledenja.

Vrstici z `[...]` sta avtorjeva sprememba in zamenjata kodo, ki se pojavi pri večini programov in obsega delovanje dinamičnega povezovalnika ter pripravo izvajalnega okolja programskega jezika C. Oba koraka skupaj obsegata okoli trideset sistemskih klicev, ki pa tukaj niso pomembni.

Sistemski klic `write` predstavlja bistvo delovanja tega programa. `strace` na pregleden način izpiše njegove argumente: `1` za datotečni deskriptor standardnega izhoda, `"test\n"` za niz, ki ga program zapiše, in `5`, ki je velikost tega niza. Njegov rezultat je `5`, in pomeni število zapisanih bajtov.

Program se zaključi s sistemskim klicem `exit_group`, ki proces in njegove niti zaključi, lupini pa vrne izhodni status. V tem primeru je to `0` in pomeni uspeh. Rezultat tega klica je `?`, saj se programi po sistemskih klicih iz družine `exit` več ne izvajajo.

Tretja pogosta uporaba sistemskega klica `ptrace` izhaja iz programov Fakeroot-ng in PRoot, ki ga uporabljata za simuliranje privilegiranega statusa neprivilegiranim procesom. Ko želi proces izvesti sistemski klic, za katerega bi običajno potreboval pravice super uporabnika, se izvede drugačna akcija, programu pa se sporoči uspeh. Na primer, ko želi program dostopati do skrbniške datoteke, mu mehanizem vstavi vsebino neke druge datoteke. Prav tako lahko spremeni rezultat funkcije `getuid` tako, da proces misli, da ima pravice super uporabnika. Oba programa se uporabljata pri gradnji programov za upravljalnike paketov distribucij (angl. *package manager*). Običajen postopek namestitve programov v operacijskem sistemu spremljajo datoteke Make z ukazi `build` in `install`, ki datoteke programa shranijo v sistemske imenike. Neprivilegirani procesi tega ne smejo početi, zato ti programi datoteke preusmerijo v običajne imenike, ki jih kasneje stisnejo in z njimi pripravijo pakete za namestitev.

Čeprav je uporaba simulacije teh dveh programov načeloma varna, v svojih priročnikih opozarjata, da nista varnostno preverjena in da je njun namen izključno uporaba pri gradnji in testiranju programov. Delovanje sistemov za gradnjo lahko v veliki meri predvidimo, prav tako pa jim praviloma zaupamo. Zaradi tega nista primerna za rešitev težave diplomskega dela, ki predpostavi možnost zlonamernih programov.

2.6.2 Omejitve in pomanjkljivosti

Sledenje je dovoljeno samo procesom istega uporabnika med seboj, prav tako pa ni mogoče slediti ostalim sledilcem ali njihovim zasledovanim procesom. Drugih omejitev privzeto ni. To je lahko težavno, saj pomeni, da lahko neprivilegirani procesi neomejeno nadzirajo ostale

procese, kar brez sistemskega klica `ptrace` ne bi bilo mogoče[15]. Posamezen proces lahko prepove sledenje nad njim s klicem funkcije `prctl(PR_SET_DUMPABLE, 0)`, kot to storijo nekateri kriptografski programi, vendar večina procesov tega ne stori, veliko pa jih ravno tako vsebuje občutljive podatke. Primer so brskalniki, ki uporabljajo podatke o uporabnikovih geslih in bančnih certifikatih, proti takim napadom pa niso zaščiteni. Izjema tem omejitvam so procesi z zmožnostjo `CAP_SYS_PTRACE` (tisti, ki jih izvaja super uporabnik), saj lahko sledijo vsem procesom ne glede na njihovo poreklo ali pripadnost uporabnikom.

Zaradi varnosti večina distribucij operacijskega sistema GNU/Linux privzeto dodatno omeji uporabo tega sistemskega klica. Omejitve se pojavijo v dveh oblikah, obe v okviru mehanizmov LSM, opisanih v razdelku 2.7. Prva je del modula Yama, ki sledenje dovoli samo nad procesi potomci. Tako ohrani možnost razhroščevanja in uporabe programa `strace` nad programi iz lupine, ne dovoli pa sledenja poljubnim že obstoječim procesom. Drugi način pride iz modula SELinux, ki prepove uporabo sistemskega klica v vseh primerih.

Za rešitev težave diplomskega dela deluje `ptrace` kot popolno orodje. Enako kot mehanizem `seccomp` (razdelek 2.5) omogoča nadzor nad vsemi sistemskimi klici, tako da program nenadzorovano ne more vplivati na ostale dele sistema. Pri tem pa vsebino vseh klicev preverjamo v lastni kodi, kar omogoča neomejeno zapletene izraze za preverjanje vseh argumentov. Prav tako uporaba klica ne zahteva pravic super uporabnika, saj se vse dogaja med procesi istega uporabnika. Dodatna prednost je, da klic implementirajo tudi drugi operacijski sistemi, osnovani na Unix, kot sta FreeBSD in Mac OS X, čeprav se delovanje njihovih različic lahko razlikuje. Edina funkcionalna omejitev je, da ni mogoče nadzorovati procesov, ki `ptrace` za svoje delovanje uporabljajo, saj takšnim ni dovoljeno slediti.

Težave se pojavijo pri poskusu implementacije takšne rešitve. Program je potrebno implementirati za vsako arhitekturo posebej, saj podatke preverjamo na nivoju registrov, dostop do pomnilnika procesa pa vršimo v velikosti celih besed (navadno 4 ali 8 bajtov), kar zahteva pozornost na pravilo debelega oziroma tankega konca. Prav tako je potrebno implementirati logiko za vseh 306 sistemskih klicev, ki so na voljo od različice jedra 3.17.6 naprej, ali pa delovanje nekaterih zanemariti. To zahteva obširno znanje o delovanju vsakega sistemskega klica na nivoju notranjega delovanja jedra.

Hujše težave so varnostne narave. Pojavi se podobna nerodnost s preverjanjem zapletenejših argumentov kot pri mehanizmu `seccomp`, kjer lahko ločena nit zlonamernega procesa tik pred izvedbo sistemskega klica njegove argumente spremeni, kar se zgodi za tem, ko smo podatke že preverili in klic odobrili. Težava pa je tukaj prisotna v obe smeri, saj nekateri sistemski klici svoje rezultate zapišejo v pomnilniški prostor procesa (primer sta `accept` in `recvmsg`), ki sistemski klic izvede. Te rezultate je včasih potrebno preveriti preden jih proces vidi, saj lahko vsebujejo občutljive podatke, kot so podatki z omrežja. Možna rešitev bi bila zaustavitev vseh niti procesov pred izvedbo sistemskega klica, vendar zaradi tveganih stanj pri sočasnih nitih za to še ne obstaja uspešna implementacija, prav tako pa bi takšna rešitev drastično upočasnila delovanje programa.

Druga možnost bi bila zapletena procesna arhitektura, podobna tisti, ki jo ima v prihodnosti namen uporabiti brskalnik Chromium s pomočjo mehanizma `seccomp` (razdelek 2.5). Takšna arhitektura bi enako kot Chromiumova vključevala dva procesa, nadzornega in omejenega, kjer bi nadzorni proces sprejemal odločitve o dovoljenju izvajanja posameznih sistem-

[15] Pomnilnik posameznega procesa je sicer na voljo tudi v datoteki `/proc/PID/mem`, ki pa jo lahko bere samo proces sam ali procesi super uporabnika.

skih klicev omejenega procesa. Rešitvi bi se razlikovali v načinu sledenja procesu. Z uporabo sistemskega klica `ptrace` ni potrebno spremeniti kode programa, da nadzorujemo njegovo delovanje, prav tako pa nam noben sistemski klic ne more uiti [51].

Poleg varnostnih težav se spopademo tudi s težavo hitrosti. Za vsak sistemski klic procesa se mora več krat zamenjati kontekst delovanja računalnika (angl. *context switch*)[16]. Najprej poskusi zasledovani proces izvesti sistemski klic, kar preklopi kontekst v jedro. Nato jedro obvesti sledilca, kar ponovno preklopi kontekst. Ko sledilec dovoli nadaljnje izvajanje, se kontekst ponovno preklopi v jedro, ki sistemski klic izvede, in njegov rezultat sporoči sledilcu. Zatem sledita še dva preklopa konteksta, iz sledilca jedru in končno iz jedra zasledovanemu procesu. To je šest preklopov konteksta, brez sledenja pa bi bila samo dva. Samo zaradi tega lahko pričakujemo približno trikratno upočasnitev vseh programov, ki pogosto uporabljajo sistemske klice. To vsebuje vse programe, ki redno komunicirajo z okoljem, kar vključuje premikanje oken v grafičnem vmesniku, predvajanje zvokov, branje podatkov iz kamer in mikrofonov, vso uporabo trdih diskov in podobno. Težava številnega preklapljanja konteksta se pojavi tudi pri uporabi funkcij za branje in pisanje v procesov delovni pomnilnik, saj lahko prenose vršimo samo v velikosti ene besede, in jih zaradi tega pri večjih količinah podatkov potrebujemo veliko.

Tudi če bi varnostne in hitrostne težave odpravili, bi implementacijo otežilo dejstvo, da je programski vmesnik sistemskega klica `ptrace` zelo neprijazen za uporabo. Na to kažejo številne pritožbe uporabnikov [46], sledi pa nekaj primerov.

1. Pogosto želimo iz pomnilnika zasledovanega procesa podatke prenašati v velikosti enega bajta. Zaradi zasnove vmesnika moramo biti pri tem pozorni na pravili debelega in tankega konca, saj so prenosi možni le v velikosti cele besede.
2. Vmesnik ne loči med procesi in nitmi, ustvarjanju in zaključevanju niti pa je potrebno slediti ročno. Zanesljivo tega ni mogoče implementirati, saj se v nekaterih primerih sporočilo o zaključku niti ne pojavi. Ko sporočilo je na voljo, je v obliki enega 32 bitnega celega števila, za njegovo interpretacijo pa so na voljo številni makroji, katerih pomen je razpršen po različnih priročnikih ki skupaj ne pokrijejo vseh primerov.
3. Vsaka operacija klica `ptrace` je lahko ne glede na prejšnje priprave v zapletenih primerih neuspešna, zato je potrebno vedno preveriti statuse vseh klicev. Prav tako je potrebno voditi evidenco dosedanjih klicev, saj se pomen statusov spreminja glede na prejšnje klice. Pri nekaterih funkcijah se rezultat klica prekriva s podatkom o napaki (na primer, če pomnilniška lokacija procesa vsebuje vrednost `-1`, koda za napako pa je prav tako `-1`), zato moramo v teh primerih dodatno preveriti vrednost globalne spremenljivke `errno`.

Razvijalci operacijskega sistema FreeBSD so mnenja, da je sistemski klic `ptrace` pomemben sestavni del sistema, zato njihova različica vključuje funkcionalne nadgradnje in popolnejšo dokumentacijo [9]. Nadgradnje vsebujejo dodatne funkcije, s katerimi lahko natančneje preverimo stanja zaustavitve in nastanka niti, ki so sicer dvoumna. Poleg tega odstranijo omejitev branja in pisanja pomnilnika v podatkih velikosti ene besede, saj nove funkcije omogočajo branje in pisanje podatkov poljubnih velikosti.

Pristop operacijskega sistema Solaris se bistveno razlikuje. V njem morajo takšno funk-

[16] Menjava konteksta delovanja (angl. *context switch*) pomeni shranjevanje stanja procesa v tako obliko, da lahko kasneje nadaljuje z izvajanjem. To pomeni shranjevanje vseh programskih registrov, prav tako pa povzroči ponastavitev predpomnilniških tabel navideznega pomnilnika.

cionalnost programi implementirati z dostopom do posebnih datotek iz imenika `/proc/PID`. Datoteke v njem predstavljajo registre procesov, vsebino njihovega pomnilnika in nadzorne datoteke, ki ob pisanju spremenijo stanje procesa. Zaradi združljivosti za nazaj je še vedno na voljo funkcija `ptrace`, ki pa ni več sistemski klic, ampak je implementirana z dostopi do omenjenega imenika. Njeno uporabnost je omejena, saj ne podpira vseh zmožnosti izvornega sistemskega klica.

Iz zgoraj opisanih razlogov `ptrace` ni primerno orodje za rešitev težave diplomskega dela. Poleg opazne upočasnitve nadzorovanih programov je implementacija rešitve z njim težavna, saj je njegova uporaba v okviru varnosti zelo zapletena, hkrati pa zahteva veliko razumevanja notranjega delovanja operacijskega sistema.

2.7 Linux Security Modules (LSM)

Linux Security Modules (LSM) oziroma moduli za varnost so ogrodje, ki omogočajo implementacijo različnih varnostnih modelov v jedru Linux. Na ta način se razvijalci jedra izognejo favoriziranju enega modela in prenesejo odločitev na uporabnike oziroma vzdrževalce distribucij. Moduli za varnost se med seboj izključujejo, tako da je lahko dejaven samo eden naenkrat, v prihodnosti pa bo ta omejitev odpravljena [5].

Ogrodje je implementirano tako, da pred dostopom do nekaterih občutljivih podatkov, kot so datoteke in podatki procesov, jedro pokliče funkcije vključenega modula LSM. V teh funkcijah lahko modul LSM izvede dodatna preverjanja in dostopa do podatkov ne dovoli. S tem se omogoči implementacija obveznega nadzora dostopa (angl. *mandatory access control* ali MAC), ki pomeni, da lahko uporabniki in njihovi procesi dostopajo samo do tistih virov, za katere so dobili izrecno dovoljenje. V obstoječem modelu nadzora dostopa lahko uporabniki uporabljajo vse vire, ki niso izrecno prepovedani, kar je ravno obratno.

Trenutno je v samem jedru na voljo pet modulov za varnost: SELinux, TOMOYO Linux, AppArmor, Smack in Yama, obstajajo pa še drugi, ki jih je potrebno v jedro dodati ročno. Večina modulov poleg kode v jedru vključuje še množico uporabniških programov za krmiljenje njihovega delovanja.

SELinux (*Security-Enhanced Linux*) je bil od modulov za varnost prvi implementiran, njegov namen pa je bila vključitev obveznega nadzora dostopa v varnostni model Linuxa. Ker so bili nekateri razvijalci mnenja, da je njegov poseg v jedro prevelik, druga pa se niso strinjali z njegovim varnostnim modelom, je bilo kot kompromis zasnovano ogrodje LSM, ki omogoča vključitev alternativnih varnostnih modelov.

SELinux nadzor nad dostopom implementira z dodatnimi atributi na imenikih in datotekah, ki jih shrani v datotečni sistem. Zaradi tega je začetna nastavitev modula zahtevna in časovno potratna, saj je potrebno nastaviti atribute vseh imenikov in datotek, katerih dostop želimo nadzirati. Poleg tega ne deluje z datotečnimi sistemi, ki ne podpirajo razširjenih datotečnih atributov (angl. *extended file attributes*). Najbolj priljubljeni datotečni sistemi podporo imajo, na primer ext2, ext3 in ext4 ter xfs in btrfs, vendar posebni sistemi kot je NFS (***N**etwork **F**ile **S**ystem*), datotečni sistem za dostop do datotek preko omrežja, z njim ne delujejo. Omogoča tudi varnostne mehanizme, ki so del njegovih sistemskih nastavitev, kot je prepoved uporabe sistemskega klica `ptrace`. Ta modul za varnost privzeto vključita distribuciji GNU/Linux Fedora in Red Hat Enterprise Linux. Uporabljata ga za utrditev sistema proti ranljivostim, ki še niso bile odkrite. Na primer, spletnim strežnikom, kot je Apache, lahko delovanje omejita na

uporabo imenika `/srv/www`, tako da v primeru vdora preko njih napadalec ne more dostopati do ostalih datotek na sistemu.

Modul AppArmor je neposredna alternativa modulu SELinux, saj ima enak funkcionalen namen. Razlikuje se v enostavnejšem skrbništvu in namestitvi, saj namesto atributov v datotečnem sistemu uporablja centralne nastavitvene datoteke, kjer za nastavitev omejitev dostopa do posameznih imenikov in datotek uporabimo poti do njih. Za dodatno olajšanje namestitve je na voljo orodje, ki tekoče programe opazuje in za njih ustvari varnostne profile (nastavitvene datoteke), ki dovolijo samo dejavnosti iz časa opazovanja programov. Zaradi uporabe osrednjih nastavitev ima nekoliko manjšo varnost od modula SELinux. Če obstajajo trde povezave[17] do datotek, ki jih napadalec ne bi smel uporabljati, morajo biti v nastavitvah tudi te navedene, saj se njihove poti razlikujejo od poti izvornih datotek. V praksi se to ne izkaže kot težava, zato distribucije, ki niso namenjene samo večjim podjetjem ampak tudi samostojnim uporabnikom, raje uporabljajo AppArmor kot SELinux. Privzeto ga uporabljata OpenSUSE in Ubuntu, ki zraven vključita prednastavljene varnostne profile za najpogosteje uporabljene programe. Ti so na primer strežnik za tiskalnike CUPS, strežnik za podatkovno bazo MySQL, program za pregledovanje dokumentov Evince in brskalnik Firefox.

TOMOYO Linux prav tako kot AppArmor daje prednost preprostosti namestitve in tudi nudi orodje za samodejno ustvarjanje nastavitev. Poleg višje varnosti z uporabo obveznega nadzora dostopa pa vključuje orodja za analizo delovanja sistema, in na tak način uporabi mehanizme LSM za nekaj, čemur niso bili namenjeni. TOMOYO Linux izhaja iz časa pred ogrodjem LSM, ko je obstajal v obliki številnih izbirnih nadgradenj za jedro. Ker ogrodje LSM ni dovolj prilagodljivo, da bi omogočilo implementacijo vseh njegovih zmožnosti, še vedno obstaja različica modula v takšni obliki.

Modul Smack (***S**implified **M**andatory **A**ccess **C**ontrol **K**ernel*) obstaja z namenom preprostosti, kot je razvidno iz njegovega polnega imena. Zaradi tega ga uporabljajo predvsem manjši sistemi, kot so Philips televizije in operacijski sistem za mobilne naprave Tizen. Modul omogoča podmnožico funkcionalnosti modula SELinux in je prav tako zasnovan na uporabi razširjenih datotečnih atributov.

Zadnji in najpreprostejši modul za varnost je Yama. Trenutno ima samo eno zmožnost, kar je omejitev sistemskega klica `ptrace` (razdelek 2.6). Njegovo delovanje prilagajamo s pisanjem števila v datoteko `/proc/sys/kernel/yama/ptrace_scope` oziroma s sistemskim klicem `sysctl`, ki je druga možnost za spreminjanje nastavitev dostopnih v imeniku `/proc/sys`. Klic je možno omejiti na štiri načine, kar sovpada s števili od `0` do `3` v omenjeni datoteki. Z `0` se privzeto delovanje ne spremeni, z `1` dovolimo uporabo klica samo nad procesi, ki so potomci klicatelja, z `2` je klic dovoljen samo privilegiranim procesom (super uporabniku) in procesom, ki sledenje izrecno dovolijo s klicem `ptrace(PTRACE_TRACEME)`. Zadnja možnost je število `3`, ki popolnoma onemogoči uporabo sistemskega klica, prav tako pa onemogoči nadaljnje spreminjanje vrednosti. Izmed vseh modulov za varnost je Yama edini, ki ga je možno uporabljati sočasno z ostalimi. Privzeto ga vključi distribucija Arch Linux z načinom omejitve `1`.

Moduli za varnost so uporabno orodje za utrjevanje sistema, ki ga lahko uporabljajo skrbniki proti zlorabam napak v programih. Ne omogočajo pa natančnega omejevanja dostopa do vseh sistemskih virov. Prav tako je njihovo delovanje po začetni namestitvi težko spremeniti,

[17] Trda povezava (angl. *hard link*) deluje enako kot simbolična povezava (»bližnjica«), s to razliko, da jo datotečni sistemi obravnavajo kot pravo datoteko. Zaradi tega programi, ki računajo zasedenost pomnilnika, datoteke s trdimi povezavami upoštevajo več krat.

saj je njihov namen predčasno utrjevanje sistema s strani skrbnikov, in tako njihovega delovanja ne moremo programsko prilagajati novim, neznanim programom. Iz teh razlogov niso uporabni za rešitev težave diplomskega dela.

2.8 Strojna virtualizacija

Strojna virtualizacija je simulacija strojne opreme celega računalnika znotraj drugega računalnika (*gostitelja*), običajno fizičnega. Simulirane strojne komponente vsebujejo centralne procesne enote, delovni pomnilnik, osnovni vhodno izhodni sistem (BIOS ali angl. ***B**asic **I**nput **O**utput **S**ystem*) in ostale. Z njo lahko znotraj tekočega operacijskega sistema poženemo ločen neodvisen računalnik (*gost*), ki ima svoj pogled na strojne naprave in vsebuje svojo programsko opremo.

Glede na podporo programske opreme ločimo različne vrste virtualizacije. Lahka virtualizacija, ki jo omogoča programski paket LXC (razdelek 2.10), je omejena na izvajanje programov za jedro določenega operacijskega sistema. Na drugačen način je omejena virtualizacija aplikacij (angl. *application virtualization*), ki dovoli samo izvajanje ožjega nabora programske opreme. Takšen pristop uporablja program Wine (***W**ine **i**s **n**ot an **e**mulator*), s katerim lahko veliko programov za operacijski sistem Microsoft Windows izvajamo na različnih izvedenkah operacijskega sistema Unix. Deluje tako, da na novo implementira standardne knjižnice operacijskega sistema Windows z uporabo funkcionalnosti, ki so na voljo v gostiteljevem operacijskem sistemu. Poleg tega vsebuje proces, ki prevzame naloge Windowsovega jedra NT, imenovan wineserver. Ta proces ponuja programom sistemske storitve, ki so jim v domačem okolju običajno na voljo, prevede signale Unix v izjeme Windows ter ukaze za grafični izris posreduje strežniku X, ki je najbolj priljubljen grafični strežnik v distribucijah GNU/Linux. Wine še ne podpira vseh zmožnosti operacijskega sistema Windows, zato z njim ne moremo izvajati vseh programov, je pa kljub temu zelo priljubljeno orodje. Razvijalci podporo običajno izboljšajo po potrebi, ko uporabniki zaprosijo za delovanje določenih programov.

Prednost strojne virtualizacije je to, da omogoča izvajanje drugih operacijskih sistemov brez spremembe njihove implementacije. Zato lahko tudi vse programe za te operacijske sisteme izvajamo brez sprememb, ti pa na videz delujejo enako kot na pravi namestitvi sistema. Prav tako nismo omejeni na programe in operacijske sisteme, ki delujejo z naborom ukazov našega procesorja, saj nekateri navidezni stroji podpirajo simulacijo nabora ukazov. V tem primeru nam to prihrani potrebo po dodatni strojni opremi. Njihova slabost pa je učinkovitost, saj simulacija na strojnem nivoju močno upočasni delovanje programov.

Navidezni stroji za strojno virtualizacijo se uporabljajo za različne namene. Pogosto jih uporabljamo za izvajanje programov, ki nimajo različice za naš operacijski sistem, saj drugega operacijskega sistema ne želimo namestiti. Prav tako lahko razvijalci z njimi takšne programe razvijajo in preizkušajo. Razvijalcem so prav posebno uporabni tisti navidezni stroji, ki omogočajo izvajanje programov za druge nabore ukazov. Takšne strojne arhitekture so lahko tudi že zastarele, tako da v fizični obliki računalniki z njimi niso več dobavljivi. Razvoj programov in operacijskih sistemov za njih je najbolj zanimiv iz raziskovalnih razlogov, saj so njihovi nabori ukazov običajno bolj enostavni od sodobnih in zato bolj primerni za učenje konceptov razvoja operacijskih sistemov. S takim pristopom lahko programiramo tudi za nabore ukazov, ki so prototipi in strojne implementacije še nimajo.

Uporaba navideznih strojev pa je pogosta tudi v strežniških okoljih, kjer več navideznih strojev teče na enem fizičnem računalniku. Prednosti takšnega pristopa so številni. Med glav-

nimi je boljši izkoristek energije in strojne opreme, saj lahko porazdelitev navideznih strojev na fizične računalnike optimiziramo tako, da manj zahtevne navidezne stroje namestimo skupaj na en fizični računalnik. Zaradi tega potrebujemo fizičnih računalnikov manj, ti pa imajo bolje izkoriščeno strojno opremo. Za uporabo tega pristopa ni potrebno, da vnaprej poznamo zahtevnost različnih navideznih strojev, saj lahko navidezni stroj preselimo na drug fizični računalnik brez motnje delovanja. Temu rečemo *migracija* ali včasih *teleportacija*. Prav tako je vzdrževanje navideznih strojev enostavnejše od fizičnih, saj vzdrževalec ne rabi posegati v strojno opremo računalnikov. Brez poseganja v strojno opremo lahko nove navidezne stroje tudi dodaja. Poleg tega navidezni stroji olajšajo okrevanje po katastrofah (angl. *disaster recovery*), saj je hranjenje njihovih varnostnih kopij relativno enostavno. V primeru naravne katastrofe ali podobnega dogodka lahko skrbniki spletni promet uporabnikov samo preusmerijo na drugo spletno mesto, kjer so varnostne kopije pripravljene na uporabo.

Najbolj znani implementaciji strojne virtualizacije sta VirtualBox in VMware Workstation, ki sta obe na voljo za operacijske sisteme GNU/Linux, Mac OS X, Microsoft Windows in različne izvedenke BSD. Oba programa za optimizacijo delovanja uporabljata posebne zmožnosti strojne opreme, ki jih omogočajo novejši procesorji z naborom ukazov x86. Te vključujejo virtualiziranje enote za upravljanje pomnilnika (angl. *memory management unit*), razširitve delovanja tabel za ostranjevanje, virtualiziranje dostopa do grafičnih kartic in podobno. Med posameznimi procesorji se podpora razlikuje, najvišjo pa imajo dražja vezja namenjena uporabi v strežnikih.

VirtualBox je svoboden in odprtokoden program. Namenjen je predvsem osebni uporabi, saj ne vsebuje naprednih orodij za vzdrževanje večih sočasnih navideznih strojev ali možnosti premikanja strojev na druge fizične računalnike med delovanjem. VMware Workstation je komercialen program in takšna orodja ima, prav tako pa nudi koristno infrastrukturo za deljenje omrežnih naprav med navideznimi stroji ali ustvarjanje mostov med njihovimi omrežnimi priključki. V splošnem je tudi njegovo delovanje hitrejše, predvsem v neposrednem dostopu do strojne opreme kot je grafična kartica.

Z uporabo strojne virtualizacije bi lahko zasnovali rešitev težave diplomskega dela. Čeprav skrbništvo navideznih strojev običajno spremljajo orodja z grafičnim vmesnikom, je na voljo tudi knjižnica libvirt, ki delovanje različnih implementacij navideznih strojev abstrahira v en vmesnik in s tem ponudi enoten programski dostop za njihov nadzor. Kljub programskemu pristopu bi navidezne stroje morali pripraviti, kar vključuje namestitev ciljnega operacijskega sistema in ostale programske opreme, ki je potrebna za izvajanje programov, ki jih želimo varno izvajati. Z uporabo navideznih strojev bi bila rešitev lahko zelo splošna, saj bi lahko podprli uporabo različnih navideznih strojev z različnimi operacijskimi sistemi ali celo z različnimi nabori ukazov procesorjev. Prav tako bi bila implementacija varnostnih omejitev z naše strani nezahtevna, saj nudijo vsi strojni navidezni stroji podrobne nastavitve zmožnosti vsebovanih programov. Te zmožnosti pa so omejene na nadzor celega vsebovanega operacijskega sistema in niso možne na nivoju posameznega programa, kar bi morali pri izračunu omejitev upoštevati. Podrobnejše omejitve bi lahko uvedli z uporabo pomožnega programa znotraj navideznega stroja, za to pa bi morali uporabiti enega od ostalih varnostnih mehanizmov.

V primerjavi z ostalimi mehanizmi za omejevanje izvajanja bi bila bolj zahtevna tudi implementacija poganjanja samih programov, saj lahko z navideznimi stroji komuniciramo samo na načine, ki so sicer na voljo pri komunikaciji s fizičnimi računalniki. To so različni omrežni protokoli, kot sta sta SSH (***Sh**ell*) in Telnet. Z njima bi se na tekoč naložen navidezni stroj

povezali in z uporabo oddaljene lupine nanj prenesli program, ki ga želimo varno izvesti. Nato bi ga pognali in njegove rezultate prav tako prenesli preko navideznega omrežja. VirtualBox vsebuje pomožne programe, ki odstranijo potrebo po posebni omrežni povezavi za poganjanje programov na gostu, ampak podpirajo prenos podatkov samo v obliki standardnega vhoda in izhoda, ne pa prenosa dodatnih datotek.

Za implementacije navideznih strojev je običajno na voljo programski paket imenovan *guest additions* (dodatki za gosta), ki ga lahko na gostu namestimo in s tem pridobimo dodatne zmožnosti na strani gostitelja. Najpogosteje jih uporabniki namestijo za t. i. *seamless* (dobesedno »brez šivov«) način delovanja, ki nadgradi integracijo med gostiteljem in gostom z namiznimi zmožnostmi, kot so potegni in spusti (angl. *drag and drop*), deljenje sistemskega odložišča, samodejno prilagajanje ločljivosti zaslona in podobne. Prej omenjena funkcionalnost poganjanja programov znotraj gosta je tudi implementirana v okviru takšnega paketa. Paketi pa ne omogočajo naprednega nadzora posameznega procesa v gostujočem operacijskem sistemu.

Pri uporabi navideznih strojev za izvajanje posameznih programov naletimo na težave z učinkovitostjo. Če želimo zagotoviti neokrnjena navidezna okolja moramo za vsako izvajanje programa uporabiti svež stroj, to pa vključuje pogon celotnega operacijskega sistema, kar je časovno zahtevno. Prav tako terjajo navidezni stroji od sistema več delovnega pomnilnika, kot bi ga rešitev potrebovala, če jih ne bi uporabljali. Zato implementacija rešitve s strojno virtualizacijo ne bi mogla biti interaktivna. Uporabniki bi morali svoje programe najprej naložiti, nato pa bi navidezni stroji rezultate objavili kasneje. Da strežnika ne bi preobremenjevali, bi stroje lahko vključili čez noč.

Iz zgoraj omenjenih razlogov je zasnova rešitve, ki za varnost posameznih programov uporablja izključno navidezne stroje, neprimerna. Sami stroji zahtevajo veliko večjo količino sistemskih virov od samih programov, kar ni učinkovito. Poleg tega s strani skrbnikov še vedno zahtevajo vzdrževanje za posodobitve in dodajanje podpore novim knjižnicam in programom, kar je pri uporabi obstoječega nespremenjenega operacijskega sistema nepotrebno. Strojne navidezne stroje bi lahko uporabili v kombinaciji z ostalimi mehanizmi, da bi dodatno omejili delovanje neznanih programov, vendar so v takem primeru odveč, saj je zadostno mero varnosti mogoče implementirati brez njih. Poleg tega je možnost velika, da so naši strežniki v resnici že navidezni stroji, tako da v okviru celega sistema že zagotavljajo varnostne prednosti navideznih strojev.

2.9 Jeziki z navideznimi stroji

Če imamo izbiro nad programskim jezikom, v katerem so neznani programi napisani, lahko izberemo tistega, ki že vsebuje mehanizme za omejevanje izvajanja. Takšne omejitve najlažje vsebujejo jeziki, ki tečejo na navideznih strojih (angl. *virtual machine*), saj se pri njih med samim programom in operacijskim sistemom nahaja dodatna programska plast. Ti navidezni stroji so drugačni od tistih iz prejšnjega poglavja, saj računalnik virtualizirajo samo na nivoju enega programa in ne celega operacijskega sistema. V okviru omejevanja programov na nivoju programskih jezikov pa obstajajo tudi pristopi, ki izvajanje omejijo s prilagojenimi prevajalniki.

Pristop s prilagojenim prevajalnikom uporablja Googlova tehnologija Native Client, ali na kratko NaCl. Zasnovana je bila zaradi potrebe po izvajanju zahtevnejših programov znotraj brskalnikov oziroma kot del spletnih strani. Z njeno vključitvijo v brskalnik Chromium

(in posledično Chrome) je postala implementacija grafično intenzivnih iger v brskalniku realnost, prav tako pa je njena uporaba priporočena pri razvoju programov za operacijski sistem Chrome OS, ki je v bistvu jedro Linux z grafičnim vmesnikom brskalnika Chrome.

Tehnologija NaCl podpira programska jezika C in C++. Razlika med običajnimi C ali C++ programi in NaClovimi je ta, da morajo uporabljati drugačno standardno knjižnico, saj knjižnico glibc zamenja knjižnica Newlib, njun vmesnik pa se bistveno ne razlikuje. Razlog za tem je prepoved neposredne uporabe sistemskih klicev, ki jih NaCl zamenja z enakovrednimi zmožnostmi gostitelja, običajno brskalnika. Tako postane brskalnik tisti, ki programom zagotavlja dodeljevanje pomnilnika, komunikacijo z drugimi procesi in druge sistemske zmožnosti. Ker programska jezika C in C++ omogočata tudi uporabo zbirnega jezika, s katerim bi lahko implementirali neposredno uporabo sistemskih klicev, je prevajalnik prilagojen tako, da kodo v zbirnem jeziku analizira in implementacije sistemskih klicev ne dovoli.

Trenutno sta na voljo prilagojeni različici prevajalnikov GNU CC in LLVM, ki sta sicer najbolj priljubljena prevajalnika za te jezike v izvedenkah operacijskega sistema Unix. Kot že rečeno dovolita samo uporabo prilagojenih knjižnic in ne dovolita uporabe kode v zbirniku, ki bi poskušala neposredno uporabljati sistemske klice. Od neprilagojenih različic se razlikujeta tudi po tem, da njun rezultat ni strojna koda temveč nizkonivojski vmesni jezik, ki se tik pred izvajanjem dokončno prevede v strojni jezik. Na ta način lahko NaCl podpira izvajanje na različnih strojnih arhitekturah brez ponovnega prevajanja, zaenkrat pa podpira x86, ARM in MIPS. Enako omogočata jezika Java in C#, vendar njuna vmesna koda ni nizkonivojska in za izvedbo zahteva obsežnejše navidezne stroje.

Brskalnik Firefox ima drugačen pristop do izvajanja učinkovite nizkonivojske kode. Za ta namen je vpeljal Asm.js, ki je stroga podmnožica jezika JavaScript, v katero lahko prevajamo druge jezike. Velika prednost pristopa je to, da izvajanje JavaScripta podpirajo vsi sodobni brskalniki, posledično pa podpirajo tudi izvajanje kode Asm.js. Višjo hitrost omogoča zaradi enostavnosti jezika, ki poenostavi prevod v hitro strojno kodo. Prav tako lahko brez prekinitve združljivosti za nazaj ta standard kadarkoli podprejo ostali brskalniki brez večjih arhitekturnih sprememb.

JavaScript je danes najpogostejši jezik z navideznim strojem. Vsi spletni brskalniki ga uporabljajo za dinamično spreminjanje spletnih strani, nekateri pa tudi za programiranje razširitev in aplikacij. Prav tako ga uporablja spletni strežnik Node.js, ki ima vedno več privržencev. Delovanje programov v JavaScriptu je mogoče strogo omejiti, ker je jezik tolmačen. Tolmač opravi vse dostope do sistemskih virov namesto programov, kar vključuje vse od dodeljevanja delovnega pomnilnika in uporabe datotečnega sistema do komunikacije preko omrežnih priključkov in dostopa do strojnih naprav.

Čeprav je uporaba JavaScripta in pristopov kot je NaCl zelo varna, in bi lahko bila primerna za rešitev težave diplomskega dela, so njihove omejitve preveč stroge. Rešitev dela zahteva zmožnosti kot so branje, ustvarjanje in spreminjanje datotek, implementacijo večnitnih programov, signaliziranje ostalih procesov in podobno, takšnih posegov pa ne dovolijo.

Drugačno vrsto jezikov z navideznimi stroji predstavljajo tisti, ki programe najprej prevedejo v prenosljiv vmesni jezik, neodvisen od arhitekture, nato pa prevedene programe na ciljnih računalnikih izvajajo njihovi navidezni stroji. Takšni so programski jeziki, ki tečejo na navideznem stroju JVM (***J**ava **V**irtual **M**achine*) ali na navideznem stroju CLR (***C**ommon **L**anguage **R**untime*), ki ga uporablja ogrodje .NET. Po samih zmožnostih sta si navidezna stroja zelo podobna. Razlikujeta se predvsem v ideologiji: JVM je uradno na voljo za skoraj vsako

kombinacijo strojne arhitekture in operacijskega sistema, ki si jo lahko izmislimo, medtem ko je .NET zaenkrat uradno podprt samo v Microsoftovih okoljih, čeprav njegova svobodna in odprtokodna različica Mono ni zanemarljiva. Po drugi strani je bil .NET od začetka zasnovan kot ogrodje za različne programske jezike, kot sta Visual Basic in C#, JVM pa primarno za programski jezik Java. Kljub temu obstaja veliko jezikov s podporo za JVM, na primer Clojure (dialekt Lispa), Groovy, Scala, JRuby (implementacija jezika Ruby), Jython (implementacija jezika Python) in številni drugi.

Oba navidezna stroja omogočata podroben nadzor dostopa do sistemskih virov. Ker sta si po delovanju tako podobna, bo podrobnejši opis v nadaljevanju osredotočen samo na programski jezik Java na navideznem stroju JVM.

Vsi javanski programi so sestavljeni iz razredov, saj jezik ne podpira kode, ki ni vsebovana v nek razred. Zaradi tega je tudi vstopna točka v program, metoda `main`, del nekega razreda. Takšen pristop pri programskih jezikih ni pogost, saj je vstopna točka običajno prosta funkcija. Če je vsa koda del razredov je zgradba vseh programov in knjižnic enotna, kar je v tem primeru prednost. Ko poskuša javanski program prvič uporabiti nek razred, se pokličejo metode razreda `ClassLoader` za nalaganje novih razredov v navidezni stroj. Njegova privzeta implementacija išče nove razrede (datoteke s končnico `.class`) v imenikih standardnih knjižnic in v ostalih prednastavljenih imenikih. Mehanizem je popolnoma prilagodljiv, saj lahko vstavimo svojo poljubno implementacijo. Ta lahko nove razrede pridobi na kakršenkoli način, na primer s prenosom z omrežja ali z iskanjem po nepredvidenih imenikih, podtakne pa lahko tudi nek drug razred ali si ga celo izmisli (ga na mestu implementira). Če razreda ne najde ali pa njegove uporabe noče odobriti, javi napako. Tako lahko javanskim programom onemogočimo uporabo poljubnih knjižnic in s tem uvedemo varnostni mehanizem.

Neodvisno od funkcionalnosti razreda `ClassLoader` je na voljo razred `SecurityManager`, ki programom znotraj navideznega stroja omejuje uporabo občutljivih sistemih virov. Podobno kot razred `ClassLoader` je vedno aktiven samo en njegov primerek istočasno. Viri pod njegovim nadzorom vsebujejo datotečni sistem, omrežne priključke, sistemsko odložišče (angl. *clipboard*), sliko na uporabnikovem zaslonu, dostop do zasebnih polj in metod drugih objektov, nadzor delovanja navideznega stroja, zamenjavo aktivnega primerka razredov `ClassLoader` in `SecurityManager` ter številne druge. Njegova privzeta implementacija spremeni delovanje glede na izvor programske kode. Ta je lahko na računalniku nameščena lokalno ali pa izvira s spleta. Prav tako razlikuje med običajnimi programi in t. i. appleti, ki so se včasih uporabljali v brskalnikih. Lokalno nameščenim programom nastavi bolj ohlapne omejitve kot ostalim. Tudi implementacijo tega razreda lahko zamenjamo s svojo in lahko s tem nastavimo poljubne omejitve dostopa do sistemskih virov.

Uporaba lastnih implementacij razredov `ClassLoader` in `SecurityManager` skupaj ni zapletena. Da z njima poženemo poljuben javanski program so potrebni naslednji koraki.

1. Ustvarimo novo nit, v kateri bo tekel javanski program, ki ga želimo omejiti. Ostali koraki postopka se izvajajo v kodi nove niti.

   Za novo nit se priporoča dedovanje razreda `InheritableThreadLocal` namesto običajnega `Thread`, saj implementira lokalno nit, ki ne deli varnostnega konteksta z zunanjo. Na ta način z zamenjavo razredov za varnost ne bomo vplivali na ostale niti.

2. Inicializiramo objekt svoje implementacije razreda `ClassLoader`, `mojClassLoader`.
3. Enako storimo s svojo implementacijo razreda `SecurityManager` in ustvarimo objekt

`mojSecurityManager`, nato pa vključimo njegove varnostne omejitve za tekoč program s klicem

```
System.setSecurityManager(mojSecurityManager);
```

Ta klic varnostne omejitve vključi tudi vsem ostalim nitim programa, vendar ne v primeru, ko uporabljamo lokalne niti.

4. Z objektom `mojClassLoader` naložimo kodo programa, ki ga želimo omejiti, in pokličemo njegovo metodo `main`, kar storimo tako:

```
mojClassLoader.loadClass(imeOmejenegaPrograma)
    .getMethod(”main”, String[].class)
    .invoke(null, (Object)params);
```

Omejitev teh razredov je, da z njimi ni mogoče omejiti količine uporabe sistemskih virov. Manjka možnost omejitve delovnega pomnilnika, ki je programom na voljo, prav tako pa ne moremo omejiti količine in hitrosti zapisov na vhodno izhodne naprave in jih zaščititi pred preobremenitvijo. Pomnilnik bi sicer lahko omejili tako, da bi zagnali še en ločen navidezni stroj, saj je omejitev mogoče nastaviti kot zastavico v ukazni vrstici, vendar za omejitev dostopa do vhodno izhodnih naprav take možnosti nimamo.

Celostna rešitev v tej smeri bi morala obsegati še časovno omejitev izvajanja. To bi lahko merila in nadzirala zunanja nit, predhodnica niti z omejenim programom. Težava pa se pojavi pri ustavljanju niti, saj za to obstaja samo metoda `Thread.stop`, ki je zastarela. V mnogih sodobnih jezikih takšna metoda ni več na voljo, ker izvajanje niti nemudoma prekine in s tem ne dovoli, da ta za sabo počisti. Zaradi tega ostanejo datotečni deskriptorji odprti in omrežne povezave vzpostavljene, kar lahko vodi do pomanjkanja pomnilnika. Tudi ta primanjkljaj pa lahko odpravimo z uporabo dodatnega primerka navideznega stroja.

S temi tehnologijami bi rešitev težave diplomskega dela lahko implementirali, a le če bi olajšali zahteve rešitve. Rešitev bi delovala samo za nekatere programske jezike, v nekaterih primerih pa bi zahtevala izvorno kodo programov za ponovno prevajanje ali celo njihovo prilagoditev, da lahko v omejenem okolju tečejo. Prav tako ne bi omogočala popolnega nadzora nad uporabo sistemskih virov. Imela bi eno prednost, in sicer to, da bi delovala na vseh strojnih arhitekturah in operacijskih sistemih, ki jih navidezni stroj izbranega programskega jezika podpira. Pri drugih možnostih je omenjene pomanjkljivosti mogoče odpraviti, zato so boljša izbira.

2.10 Linux containers (LXC)

Linux containers (LXC) je implementacija vsebnikov za lahko virtualizacijo, ki je grajena na mehanizmih za omejevanje procesov jedra Linux. Vsebniki povezujejo tehnologijo nadzornih skupin (razdelek 2.4) za upravljanje s sistemskimi viri, s tehnologijo imenskih prostorov (razdelek 2.3) za izolacijo sistemskih virov. Kot pomožni mehanizem pa uporabljajo tudi sistemski klic `chroot` (razdelek 2.2).

Lahka virtualizacija je vrsta virtualizacije, ki ne simulira strojne opreme ali operacijskih sistemov, ampak izolira skupine procesov od ostalih procesov na sistemu, vsi pa tečejo na istem jedru. Izolira jih do te mere, da se z njihovega vidika nahajajo na lastnem računalniku, saj si lastijo svoje primerke sistemskih virov, z drugimi procesi na sistemu pa ne morejo neposredno komunicirati. Še vedno lahko komunicirajo na enak način kot ločeni fizični računalniki, z

vzpostavljanjem omrežnih povezav, a le, če jim omogočimo uporabo omrežnih naprav.

LXC vsebuje cel programski paket, s katerim lahko z vsebniki upravljamo. Vsa orodja so v obliki ukazov za lupino, na voljo pa je tudi programski vmesnik za programski jezik C. Z ukazi lahko ustvarimo stalne vsebnike, ki so podobni primerkom strojnih navideznih strojev in vsebujejo vse datoteke celega operacijskega sistema in njegovega programja. Ko tak stroj zaženemo se incializira kot običajen operacijski sistem, vendar brez zagonskega nalagalnika in brez nalaganja samega jedra, saj sta se ta dva koraka že zgodila v okviru gostujočega računalnika. Preostali korak je zagon inicializacijskega programa, običajno imenovanega `init`, ki nastavi datotečni sistem, požene procese za storitve (demone) in omogoči prijavo uporabnika. Poleg stalnih vsebnikov obstajajo tudi spremenljivi (angl. *volatile*) vsebniki, ki jih LXC ustvari po potrebi, ko želimo v omejenem okolju izvesti posamezen program, zanj pa nimamo pripravljenega stalnega vsebnika.

Vsak vsebnik deluje glede na nastavitve iz nastavitvene datoteke. V tej lahko podrobno nastavimo delovanje nadzornih skupin in imenskih prostorov, ki jih bo vsebnik uporabljal, podpirajo pa tudi druge zmožnosti. Privzeto (s prazno nastavitveno datoteko) LXC za vsebnik ustvari novo nadzorno skupino, nove imenske prostore PID, nove imenske prostore IPC in nove imenske prostore priklopov. Teh varnostnih mehanizmov ne moremo izključiti, drugih pa privzeto ne uporablja. To pomeni, da bodo procesi znotraj vsebnika še vedno uporabljali omrežne naprave in podatke UTS gostitelja, prav tako pa bodo imeli dostop do izvornega datotečnega sistema. Dodatne imenske prostore LXC vključi, če navedemo nastavitve, ki jih potrebujejo. Na primer, če vnesemo nastavitve omrežnih naprav znotraj vsebnika, bo za vmesnik ustvarjen nov imenski prostor omrežij in priključkov gostitelja ne bo mogel uporabljati. Podobno se bo vključil imenski prostor UTS, če navedemo ime gostitelja. Zmožnosti, ki jih lahko v nastavitveni datoteki vključimo, so naslednje.

1. Pot do korena datotečnega sistema vsebnika. Če ga nastavimo, bodo imeli procesi znotraj vsebnika drug koren datotečnega sistema od procesov zunaj. Njegova uporaba zahteva, da imenik vsebuje vse datoteke, ki jih programi za delovanje potrebujejo. To so skupne knjižnice programov, slike ikon za programe z grafičnim vmesnikom, privzete nastavitve programov in druge datoteke, ki se običajno nahajajo v imeniku `/usr`[18]. Spremembo korenskega imenika LXC implementira z uporabo imenskih prostorov priklopov in sistemskega klica `chroot`.
2. Seznam dodatnih priklopnih točk. Z njimi lahko s povezovalnimi priklopi omogočimo procesom iz vsebnika dostop do datotek zunaj njihovega novega korenskega imenika. S tem se izognemo kopiranju datotek, omenjenih v prejšnji točki. Po drugi strani lahko izvorni korenski imenik obdržimo, nato pa s povezovalnimi priklopi skrijemo nekatere od njegovih imenikov. Na ostale procese v sistemu spremembe ne vplivajo, saj teče vsebnik v svojem imenskem prostoru priklopov. Pri uporabi dodatnih priklopnih točk pa moramo biti previdni v primeru, ko gostitelj uporablja priklope s skupnim deljenjem, saj na takšne lahko vplivamo tudi iz drugih imenskih prostorov.
3. Vključimo lahko samodejno polnjenje imenika `/dev` s posebnimi navideznimi datotekami, kot so `random` za naključna števila, `null` za posebno datoteko, ki je vedno prazna, in `zero` za datoteko z neskončnim številom ničel. Možnost lahko izključimo zato, da lahko

[18] Ime imenika `/usr` ne pomeni uporabnik (angl. *user*) ampak ***U**nix **S**ystem **R**esources.* V operacijskih sistemih Windows je enakovreden imenik `Program Files`.

vsebnik uporablja svoje incializacijske programe za ustrezno napolnitev imenika.

4. Povezovanje terminalov (TTY) z gostiteljevimi in ustvarjanje novih navideznih terminalov (angl. *pseudo terminal* ali PTY), kar omogoča uporabo terminalov gostitelja. Terminali so običajno dostopni preko tipkovnih bližnjic od `CTRL+ALT+F1` do `CTRL+ALT+F6`, navidezne terminale pa uporabljajo uporabniški simulatorji terminalov kot je X Terminal ali `xterm`. Če jih nastavimo, dobimo do vsebovanega sistema enostavnejši dostop, saj se izognemo potrebi po nastavitvi omrežja v vsebniku in uporabi orodij za komunikacijo preko omrežja.
5. Izberemo lahko signale, ki jih prejme proces s `pid` 1 znotraj vsebnika, preden vsebnik izključimo. Izberemo lahko signal za običajno ustavitev, ki procesom omogoči da zaključijo z najbolj kritičnimi opravili, in signal za prisilno ustavitev, ki mora vsebnik nemudoma uničiti.
6. Konfiguriramo lahko omrežje znotraj vmesnika. Če nastavitve navedemo, bo LXC za vsebnik ustvaril nov imenski prostor omrežij. Navedemo lahko konfiguracije za več omrežnih priključkov hkrati, vsaka konfiguracija pa ima svoj tip omrežja. Na primer `veth` iz razdelka 2.3.3 ali bolj zapletene, kot so `vlan` in `macvlan`, za ustvarjanje mostov med omrežji vsebnika in gostitelja. Vsebniku lahko s tipom omrežja `phys` priredimo tudi fizične naprave, ki zatem na gostitelju več ne bodo na voljo. Zanimivi pa sta tudi posebni možnosti `empty`, ki vsebniku ustvari nov imenski prostor omrežij brez nastavljenih naprav, in tako onemogoči uporabo omrežja, ter možnost `none`, ki novega imenskega prostora izrecno ne ustvari in s tem omogoči uporabo omrežnih priključkov gostitelja.
7. Določimo lahko ime gostitelja procesov v vsebniku. Če to nastavitev navedemo, bo LXC za vsebnik ustvaril nov imenski prostor imen UTS.
8. Nastavimo lahko delovanje vsakega od podsistemov nadzornih skupin iz razdelka 2.4. Na voljo imamo vse zmožnosti, ki so na voljo pri običajni ročni nastavitvi, kot so omejitve uporabe datotek iz imenika `/dev`, omejitev količine delovnega pomnilnika in omejitev izvajanja na določenih jedrih procesorja.
9. Vsebnikom, ki tečejo s pravicami super uporabnika, lahko onemogočimo uporabo posameznih skrbniških zmožnosti (angl. *capabilities*) [31]. Takšne so poljubno spreminjanje lastništva katerihkoli datotek, nenadzorovano pošiljanje signalov procesom, ignoriranje pravil varnostnih modulov LSM, opravljanje opravil v imenu vseh drugih uporabnikov, ustvarjanje novih imenskih prostorov, uporabljanje sistemskega klica `chroot`, nameščanje gonilnikov v jedro in ostale privilegirane operacije.
10. Delovanje vsebnika lahko dodatno omejimo z izbiro profila za AppArmor ali z izbiro konteksta za SELinux iz razdelka 2.7, če ju naše jedro podpira.
11. Vključimo lahko omejitve mehanizma `seccomp` (razdelek 2.5) z nastavitveno datoteko, ki vsebuje dovoljene in prepovedane sistemske klice ter odzive na njihovo izvedbo, če jih ne dovolimo.
12. Nastavimo lahko preslikave `uid` in `gid` pri uporabi imenskih prostorov uporabnikov (razdelek 2.3.6) v njihovem standardnem formatu.
13. Vključimo lahko izvedbo poljubnih programov in skript, ko se vsebnik premakne v določeno stanje. Na primer, pred njegovo pripravo lahko na zaslon izpišemo obvestilo, da se bo vključil. Prav tako lahko tik za ustvarjanjem imenskih prostorov na roke priredimo priklopne točke. Možnost pa imamo tudi nadgraditi delovanje samodejnega polnjenja

imenika `/dev`, če smo funkcijo vključili.

14. Nastavimo lahko atribute za samodejni zagon vsebnikov ob zagonu sistema (angl. *autostart*), ki jih uporabljajo zunanji programi. Primer programa je storitveni demon z imenom `lxc`, ki ob zagonu sistema takšne vsebnike zažene.
15. Spremenimo lahko privzeto pot do datoteke, kamor se napake in sporočila delovanja posameznega vsebnika zapišejo. Določimo lahko tudi, kako resna morajo sporočila biti, da bodo v izpis vključena. Privzeto se izpisujejo samo napake, dodamo pa lahko še opozorila, splošne informacije in podatke za razhroščevanje.
16. Nazadnje lahko izvajanje procesov omejimo na uporabo 32 bitnih programov, če ima naš računalnik procesor s 64 bitno strojno arhitekturo, kot sta x86_64 in amd64. Ta podatek uporabljajo upravljalniki paketov distribucij znotraj vsebnikov za izbiro različic programov, ki jih želimo namestiti.

Katere od teh zmožnosti so na našem sistemu na voljo je odvisno od jedra, ki ga naša sistem uporablja. Za pregled teh je na voljo pomožni program `lxc-checkconfig`, ki funkcionalnosti izpiše. To je primer takšnega izpisa iz distribucije Arch Linux:

```
$ lxc-checkconfig
— Namespaces —
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

— Control groups —
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

— Misc —
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

Iz izpisa je razvidno, da so v jedru na voljo vse zmožnosti razen imenskih prostorov uporabnikov. Arch Linux teh privzeto ne omogoča, saj so se v času od njihove uvedbe pojavile številne varnostne ranljivosti. Čeprav so bile že odpravljene, skrbniki distribucije implementaciji teh imenskih prostorov še ne zaupajo.

V primerjavi z ostalimi mehanizmi za omejevanje imajo vsebniki LXC številne prednosti. Programi v njihovih okoljih tečejo s polno hitrostjo, česar strojna virtualizacija ne omogoča, ob tem pa nudijo zadostno mero varnosti. Prav tako je za omejevanje posameznih programov njihov zagon skoraj takojšnji, saj ni potrebe po zagonu celotnega operacijskega sistema. Njihova uporaba pa je od ostalih enostavnejša, saj je integracija z gostiteljem enostavna. Zunanji

programi imajo poln dostop do datotek znotraj vsebnikov, lahko pa tudi poganjajo programe v njih. Še vedno pa je po želji na voljo komunikacija preko omrežja. Poleg tega je priprava vsebnikov preprosta, saj lahko uporabimo programe nameščene na gostiteljevem operacijskem sistemu. V primerjavi s celostno rešitvijo z uporabo sistemskega klica `ptrace` nam poleg višje hitrosti olajšajo tudi sledenje novim procesom, ki jih omejeni procesi ustvarijo, saj ti ne morejo pobegniti iz nadzornih skupin.

Imajo pa tudi slabosti, saj implementacija infrastrukture za virtualizacijo v jedru še ni dokončana. Težava se pojavi pri uporabi nadzornih skupin skupaj z imenskimi prostori, saj imenski prostori za nadzorne skupine še niso na voljo [10]. Zaradi tega morajo imeti procesi znotraj vsebnikov pregled nad vsemi nadzornimi skupinami sistema, če želijo svoje spreminjati, kar je obvezno v primeru virtualizacije celega operacijskega sistema. V primeru izvajanja enega programa v vsebniku pa to ni problem. Prav tako v okviru nadzornih skupin manjkajo nekateri podsistemi, kot je možnost omejevanja števila procesov znotraj nadzorne skupine [11]. Zaradi tega moramo pri implementacije rešitve paziti na programe, ki bi sistem želeli upočasniti z ustvarjanjem velikega števila procesov, t. i. napad *fork bomb*, imenovan po sistemskemu klicu `fork`, ki ustvari nov proces. Za izvedbo napada je dovolj, da program v neskončni zanki sistemski klic vedno znova kliče.

Za implementacijo rešitve težave diplomskega dela sami vsebniki LXC ne omogočajo vsega, kar potrebujemo. Nimajo podpore za časovno omejitev izvajanja programov, podpore za omejitev velikosti datotečnega sistema, ki je programom na voljo, ali podpore za omejitev števila procesov znotraj vsebnika. Te slabosti pa lahko odpravimo s pomožnim programom, ki ustrezen datotečni sistem pripravi in delovanje vsebnika po potrebi prekine. Zaradi preproste uporabe, visoke hitrosti in zadovoljive varnosti so vsebniki najbolj primeren mehanizem za uresničitev ciljev diplomskega dela. Najboljša rešitev, ki paketa LXC ne bi uporabljala, pa bi bila ponovna implementacija njegovih funkcionalnosti. Takšen pristop ima program `playpen` [41], ki je namenjen omejevanju posameznih programov, vendar njegova implementacija še ni tako dovršena kot vsebniki LXC.

Poglavje 3

Implementacija rešitve

V tem poglavju bomo opisali programsko rešitev težave diplomskega dela. Ta obsega program za ukazno vrstico s katerim lahko poljuben neznan program izvedemo varno, brez skrbi, da bi vplival na druge dele našega sistema. Prav tako rešitev vsebuje spletni vmesnik do tega programa v obliki sodniškega sistema za programske domače naloge. Spletni vmesnik omogoča interaktivno programiranje in izvajanje programov, kot tudi pregled oddanih rešitev in primerjanje njihovega rezultata z rezultatom referenčne implementacije.

3.1 Program `gaol`

V okviru diplomskega dela smo razvili program `gaol`, ki omogoča varno izvajanje neznanih programov v ukazni vrstici. Z njim drugim programom onemogočimo uporabo občutljivih delov datotečnega sistema in omejimo ali prepovemo dostop do pomembnih sistemskih virov. Implementiran je z uporabo varnostnih mehanizmov, ki jih nudi jedro operacijskega sistema Linux, in s pomočjo pomožnih programov, ki so zanj na voljo.

Beseda `gaol` je alternativno pisanje angleške besede jail, ki pomeni zapor. Bila je v uporabi predvsem v zgodovini, lahko pa jo še danes srečamo v časopisih iz Avstralije.

3.1.1 Cilji in izbira tehnologije

Cilj programa za rešitev težave diplomskega dela je varno izvajanje neznanih programov. Ker so neznani jim je potrebno prepovedati dostop do datotek uporabnikov in ostalih občutljivih podatkov na sistemu. Prav tako jim moramo omejiti uporabo sistemskih virov, saj bi sicer lahko sistem preobremenili in tako upočasnili izvajanje ostalih programov. Omejenim programom pa moramo zagotoviti dovolj zmožnosti, da lahko svoje delo opravijo in vrnejo uporabne rezultate.

Programom moramo omejiti naslednje sistemske vire:

1. čas izvajanja,
2. uporabo delovnega pomnilnika,
3. uporabo prostora na pomnilniških medijih,

4. število procesov potomcev, ki jih smejo ustvariti in
5. uporabo omrežnih priključkov.

Poleg omejitve sistemskih virov jim je potrebno prepovedati branje datotek ostalih uporabnikov na sistemu. Da pa je lahko njihovo delovanje uporabno moramo zagotoviti naslednje zmožnosti:

1. branje standardnega vhoda in pisanje v standardni izhod,
2. dodeljevanje delovnega pomnilnika,
3. ustvarjanje datotek, pisanje v njih in branje iz njih,
4. ustvarjanje novih procesov in niti, in
5. opcijsko uporabo omrežnih priključkov.

Program `gaol` uporabljamo kot pripomoček za varno izvajanje programov v ukazni vrstici. Posamezne omejitve nastavimo kot zastavice, ko program pokličemo. Njihove privzete vrednosti in najvišje dovoljene vrednosti pa program prebere iz nastavitvenih datotek.

Za implementacijo rešitve smo izbrali tehnologijo LXC za lahko virtualizacijo (razdelek 2.10), saj združi več mehanizmov za varnost v eno celoto in zato zmanjša zahtevnost implementacije. Prav tako je njena varnost ustrezna, omejitve pa ne upočasnijo izvajanja programov. Največja pomanjkljivost lahke virtualizacije, da zahteva uporabo istega jedra operacijskega sistema za izvajanje programov, pa v našem primeru ni slabost, saj želimo omejiti obstoječe nameščene programe, ki bi že brez omejitev tekli na tem istem jedru.

3.1.2 Implementacija

Za implementacijo programa `gaol` bi bil ustrezen vsak programski jezik z zmožnostjo klicanja sistemskih klicev in drugih opravil na nižjem nivoju. Izbrali smo programski jezik D[1], saj je od bolj priljubljenih izbir, kot sta C in C++, prijetnejši za uporabo, prav tako pa lahko ovrednotimo njegovo uporabnost za naše prihodnje programe.

Program ob zagonu najprej naloži privzete nastavitve iz nastavitvenih datotek iz imenika `/etc/gaol`, ki so podrobneje opisane v razdelku 3.1.3, in jih združi z enakovrednimi možnostmi z ukazne vrstice. Če je ali format napačen ali pa pride do napak pri branju datotek, program javi napako in prekine z delovanjem. Ob uspešnem branju nastavitev pa še preveri, ali so vnesene omejitve znotraj dovoljenih vrednosti, saj lahko v okviru njegovih sistemskih nastavitev določimo zgornje dovoljene meje omejitev, da uporabniki ne bi mogli prekoračiti zmožnosti našega strežnika.

Naslednji korak je nastavitev ravnanja s signali. Trenutna implementacija preprosto blokira prejem vseh signalov razen `SIGKILL` in `SIGSTOP`, ki ju ni mogoče ignorirati. Če signalov ne bi blokirali ali za njih ne bi nastavili drugačnih rutin za ravnanje, bi operacijski sistem naš program ob prejemu signala zaključil. Tega ne smemo dovoliti, saj naš program spremlja delovanje vsebnika LXC in ga ob prekoračenju časovne omejitve zaustavi. Če bi bil naš program prisilno ugasnjen, bi vsebnik LXC nadaljeval z delovanjem.

Zatem `gaol` spremeni lastnika svojega procesa v super uporabnika, saj večina distribucij GNU/Linux še ne dovoli uporabe imenskih prostorov uporabnikov, kar bi omogočilo implementacijo neprivilegiranih vsebnikov. Sprememba lastnika na tak način je možna le, če ima

[1] Programski jezik D sodi v skupino t. i. sistemskih jezikov, kamor sodita tudi C in C++. Omogoča približno enake funkcionalnosti kot C++, ampak jih združi v bolj sodoben in manj zapleten jezik.

datoteka s programom nastavljeno zastavico `setuid`, ki je opisana v razdelku 2.2. Zato je program pri namestitvi potrebno z zastavico opremiti.

Nato program poveča številčno vrednost v datoteki `/tmp/gaol/cur_instances` za 1, ki vsebuje število trenutnih tekočih primerkov programa. Če program ugotovi, da je primerkov preveč, javi napako in se zaključi. Najvišje dovoljeno število je zapisano v eni od nastavitvenih datotek iz prejšnjih korakov. Omejitev je na mestu za preprečevanje preobremenjevanja sistema s strani uporabnikov. Ko program z delovanjem zaključi ponovno posodobi vrednost v datoteki. Pri spreminjanju datoteke se vedno uporablja zaklepanje za sinhronizacijo, da ne povzročimo tveganega stanja.

Naslednji korak je priprava okolja za zagon vsebnika. Program ustvari imenik, ki ga bo program znotraj vsebnika lahko uporabljal za delo z datotekami. Ta imenik bo zanj predstavljal domač imenik uporabnikov (`/home`) in imenik za začasne datoteke (`/tmp`) hkrati. To je mogoče zaradi uporabe povezovalnih priklopov. Na ta imenik nato s sistemskim klicem `mount` priklopi nov začasni datotečni sistem tipa `tmpfs`, ki vsebuje tudi možnost omejitve velikosti. Na ta način `gaol` programom omeji količino obstojnega pomnilnika, ki se v resnici nahaja v delovnem pomnilniku gostitelja. Ostale imenike datotečnega sistema bo program še vedno imel na voljo, kar vključuje imenik s sistemskimi nastavitvami `/etc`. To pa je obvezno, saj nekateri prevajalniki nastavitve iz imenika potrebujejo za delovanje.

V nov imenik nato skopira dodatne datoteke, ki jih uporabnik lahko v ukazni vrstici navede. To zahteva previden prijem, saj v tem trenutku program teče kot super uporabnik. Zlonameren uporabnik bi lahko navedel pot do skrbniške datoteke in znotraj vsebnika dobil dostop do njene vsebine. Zaradi tega program začasno izpusti efektivne pravice super uporabnika in prevzame efektivne pravice tistega uporabnika, ki ga je na začetku pognal. Ko datoteke skopira zopet prevzame identiteto super uporabnika. Takšne spremembe identitete so mogoče, ker program spreminja samo *efektivne* pravice, ne pa dejanskih. To lahko storimo s sistemskima klicema `seteuid` (***set effective user identifier***) in `setegid` (***set effective group identifier***). Najprej pa je potrebno spremeniti identifikator skupine, saj ga lahko spremeni samo proces z efektivnimi pravicami super uporabnika.

Zatem program s sistemskim klicem `chown` spremeni lastnika novega imenika na tistega uporabnika, ki bo omejen program znotraj vsebnika pognal. Tega uporabnika prav tako nastavimo v eni od nastavitvenih datotek. Če tega ne bi storil, bi bile datoteke last super uporabnika in jih omejen program ne bi mogel uporabljati.

V tem trenutku imamo pripravljeno okolje za zagon vsebnika in lahko z uporabo nastavitev iz datotek in ukazne vrstice pripravimo ukaz za zagon mehanizma LXC. Vanj vključimo podatke o imenu gostitelja, nastavitvah omrežja, in omejitve delovnega pomnilnika. Ker lahko v okviru nastavitve vsebnika navedemo dodatne priklopne točke to izkoristimo tako, da prej omenjen začasni imenik preslikamo na pot `/home` in pot `/tmp` hkrati. Če bo program ta dva imenika uporabljal bo v njih videl enake datoteke.

Kot argument vsebniku podamo tudi ukazno vrstico za zagon programa, ki se bo v njegovi lupini pognal. Tega pa ne posredujemo v goli obliki, ampak ga preoblikujemo v takšno:

```
cd /home
&& ulimit -u <največje število procesov>
&& nice -n 40 sh -c <program>
```

Razlog za prvi del ukaza je to, da se bodo programi privzeto nahajali v za njih pripravljenemu domačemu imeniku in vanj shranjevali rezultate. Drugi del ukaza je posledica trenutnih ome-

jitev nadzornih skupin, ki nimajo podpore za omejitev števila procesov. S programom `ulimit` lahko takšno omejitev nastavimo vsem procesom, ki se bodo od njegovega klica naprej izvajali v lupini. V tretji vrstici s programom `nice` nastavimo prioriteto programa na najnižjo, da ne bo motil ostalih na sistemu, in ga nazadnje poženemo s privzeto sistemsko lupino `sh`.

Sledi zagon vsebnika, ki ga program opravi tako, da najprej s sistemskim klicem `fork` ustvari nov proces. Ta nov proces s sistemskim klicem `execve` svojo kodo zamenja s kodo programa `lxc-execute`, ki je namenjen izvajanju posameznega programa znotraj vsebnika LXC. Naš obstoječ nadzorni proces pa vstopi v zanko, kjer periodično opazuje stanje programa v vsebniku. Če se program v vsebniku normalno zaključi, se zaključi tudi nadzorni program in izvajanje programa `gaol` je končano. Če pa omejen program prekorači dovoljen čas izvajanja, pošlje `gaol` vsebniku signal `SIGTERM` da ga zaključi. Po pošiljanju signala nekaj časa počaka, da se ugasne. Če se vsebnik ne zaključi ali postane neodziven mu pošlje signal `SIGKILL`, ki ga zaključi prisilno.

V prihodnosti bo implementacijo programa `gaol` mogoče izboljšati, saj uporablja nekatere gradnike, ki jih bodo prihodnje različice LXC, nadzornih skupin in imenskih prostorov izboljšale. To je na primer uvedba podsistemov nadzornih skupin za omejevanje števila procesov, kar bo poenostavilo ukazno vrstico znotraj vsebnika. Podobno bo nadgradnja LXC poenostavila ustavljanje vsebnikov s pomožnim programom `lxc-stop`, ki sicer že obstaja, ampak ne deluje zanesljivo. Prav tako je v novejših različicah LXC uporaba pomožnih programov odveč, saj nudijo vmesnik za programski jezik C. Dodatna izboljšava, ki bi jo lahko implementirali zdaj, je posredovanje signalov programu znotraj vsebnika, saj jih trenutno ignoriramo. Potrebe po tej zmožnosti pa do zdaj ni bilo.

3.1.3 Navodilo za uporabo

Program `gaol` uporabljamo z ukazne vrstice. Če ga poženemo z zastavico `-h` ali `--help` izpiše svoje zmožnosti:

```
Uporaba: gaol [OPTIONS] -n NAME -c COMMAND
NAME je edinstveno ime vsebnika.
COMMAND je ukaz, ki se bo izvedel znotraj vsebnika.

Obvezni argumenti dolgih različic možnosti so prav tako obvezni pri kratkih.
OPTIONS so kombinacija:
    -f, -file=FILE      Pot do datoteke za kopiranje v vsebnik.  Možnost
                        se lahko več krat pojavi za kopiranje večih datotek.
    -s, -fssize=NUM     Velikost datotečnega sistema vsebnika v megabajtih.
    -w, -network=NUM    Ali je uporaba omrežja dovoljena (1 = da, 0 = ne).
    -p, -processes=NUM  Število dovoljenih procesov znotraj vsebnika.
    -m, -memory=NUM     Omejitev delovnega pomnilnika v megabajtih.
    -t, -time=NUM       Časovna omejitev izvajanja v sekundah.
    -h, -help           Izpis tega sporočila.

-n in -c imata dolgi različici -name in -command.

Vrne 0 v primeru uspeha in 1 v primeru napake.
```

Privzete in najvišje dovoljene vrednosti omejitev nastavimo z nastavitvenimi datotekami v imeniku `/etc/gaol`. Te so naslednje:

1. `lxc_conf` je nastavitvena datoteka vsebnika LXC, ki se mehanizmu posreduje. Z njo lahko nastavimo dodatne možnosti, ki preko argumentov ukazne vrstice niso na voljo. Na

primer seznam dovoljenih naprav v imeniku `/dev` in ostale iz razdelka 2.10.

2. `lxc_user` vsebuje ime uporabnika, v imenu katerega bo tekel program znotraj vsebnika. Uporabnik mora na sistemu gostitelja obstajati, prav tako pa mora imeti nastavljeno privzeto lupino. Zato uporabnika `nobody`, ki se pogosto uporablja za storitve, ni mogoče uporabiti. Priporoča se uporaba novega uporabnika ustvarjenega za uporabo s programom `gaol`.
3. `default_fs_size_mb` in `max_fs_size_mb` sta privzeta in najvišja dovoljena velikost vsebnikovega datotečnega sistema v megabajtih.
4. `default_mem_mb` in `max_mem_mb` sta privzeta in najvišja dovoljena količina delovnega pomnilnika, ki je programu znotraj vsebnika na voljo.
5. `default_network` vsebuje privzeto nastavitev omrežja: 1 za vključeno in 0 za izključeno.
6. `default_processes` in `max_processes` sta privzeto in najvišje dovoljeno število procesov znotraj vsebnika.
7. `default_time_s` in `max_time_s` sta privzet in najvišji dovoljen čas izvajanja programa znotraj vsebnika.
8. `max_instances` vsebuje največje dovoljeno število sočasnih primerkov programa `gaol`.

Uporaba programa `gaol` zahteva posebno previdnost na sistemih, ki za inicializacijski program uporabljajo `systemd`. Ta namreč spremeni privzeto deljenje priklopa korenskega imenika iz zasebnega na skupno. Posledica tega je, da se spremembe priklopov v omejem okolju imenskih prostorov prenesejo na korenski imenik sistema. Če pri takih nastavitvah uporabimo `gaol`, bosta domač imenik `/home` in imenik za začasne datoteke `/tmp` postala nedosegljiva vsem uporabnikom na sistemu, ne samo omejenemu programu. Težavo rešimo tako, da spremenimo nastavitev deljenja korenskega imenika na zasebno, kar lahko v ukazni vrstici storimo z naslednjim ukazom:

```
# mount --make-rprivate /
```

Razen zgoraj omenjene nevarnosti program `gaol` skrije še zadnje podrobnosti varnega izvajanja programov z vsebniki za lahko virtualizacijo, in je zato priročen za uporabo. Po začetni namestitvi si moramo samo še izmisliti enolično ime vsebnika in lahko izvajamo neznane programe na varen način.

3.2 Spletni sodniški sistem

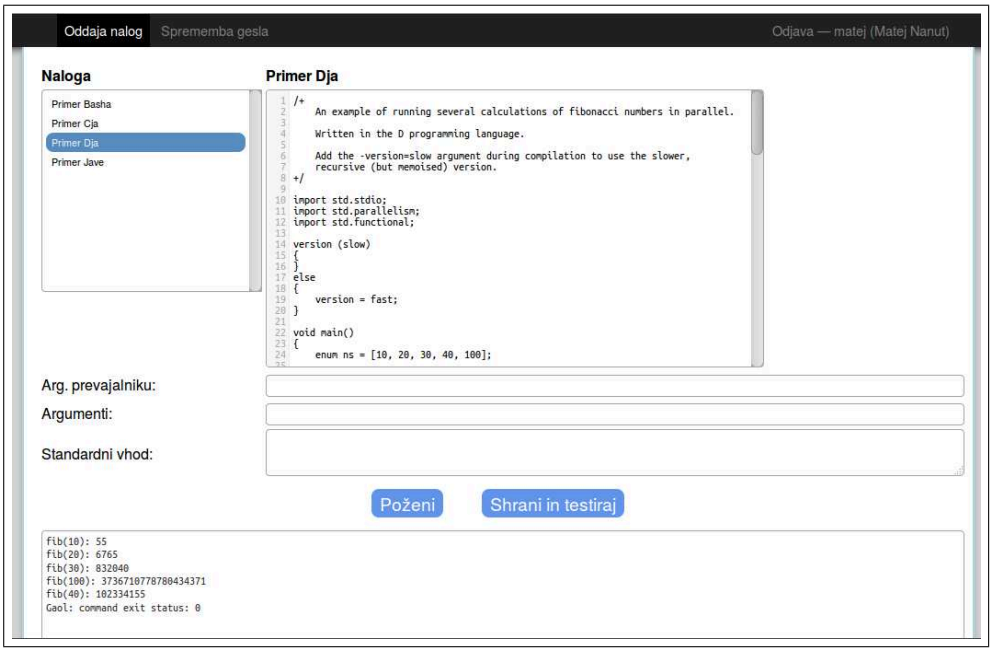
Poleg programa za ukazno vrstico smo v okviru diplomskega dela razvili tudi spletni sodniški sistem, ki je spletni vmesnik za `gaol`. Poleg varnega izvajanja programov omogoča interaktivno programiranje in preizkušanje programov, oddajanje rešitev programskih domačih nalog, primerjavo rezultatov rešitev z rezultati referenčnih implementacij in dodajanje novih nalog za različne programske jezike.

Implementiran je v programskem jeziku PHP brez uporabe dodatnih ogrodij, za hranjenje podatkov pa uporablja relacijsko podatkovno bazo SQLite. Podatkovni model vsebuje podatke o uporabnikih, domače naloge in oddane rešitve, testne primere za preizkušanje domačih nalog in rezultate preizkušanja oddanih rešitev. Prav tako vsebuje podatke o povezavah med uporabniki in skupinami nalog, ki omogočajo pregled nad tem, kdo lahko rešuje katere naloge.

V spletno stran se prijavimo z uporabniškim imenom in geslom, ki nam ju pove skrbnik. Navadnim uporabnikom so na voljo samo oddaja nalog, sprememba gesla in odjava. Skrbnik

pa vidi pregled vseh nalog z oddanimi rešitvami po uporabnikih, pregled, urejanje in preizkušanje referenčnih rešitev, seznam nalog in urejanje njihovih nastavitev, seznam uporabnikov z možnostjo brisanja in dodajanja novih ter nastavitve skupin za nadzor dostopa do določenih nalog.

Pri oddaji naloge (slika 1) lahko na levi strani okna izberemo nalogo, ki jo želimo rešiti. Ob kliku se odpre naša dosedanja shranjena rešitev. Zatem lahko v oknu interaktivno programiramo, s klikom na gumb Poženi pa svojo rešitev sproti preizkušamo. Na voljo imamo tudi okna za vnos argumentov programu in standardnega vhoda. Prav tako pa lahko sami izberemo argumente prevajalnika. Rešitev se v sistem shrani komaj, ko stisnemo gumb Shrani in testiraj. V tem trenutku pa se samodejno poženejo tudi vsi testni primeri, ki so za nalogo določeni. Pri testnih primerih se naši vnosi argumentov prevajalniku, argumentov programu in standardnega vhoda in upoštevajo, saj navede vsak testni primer svoje.

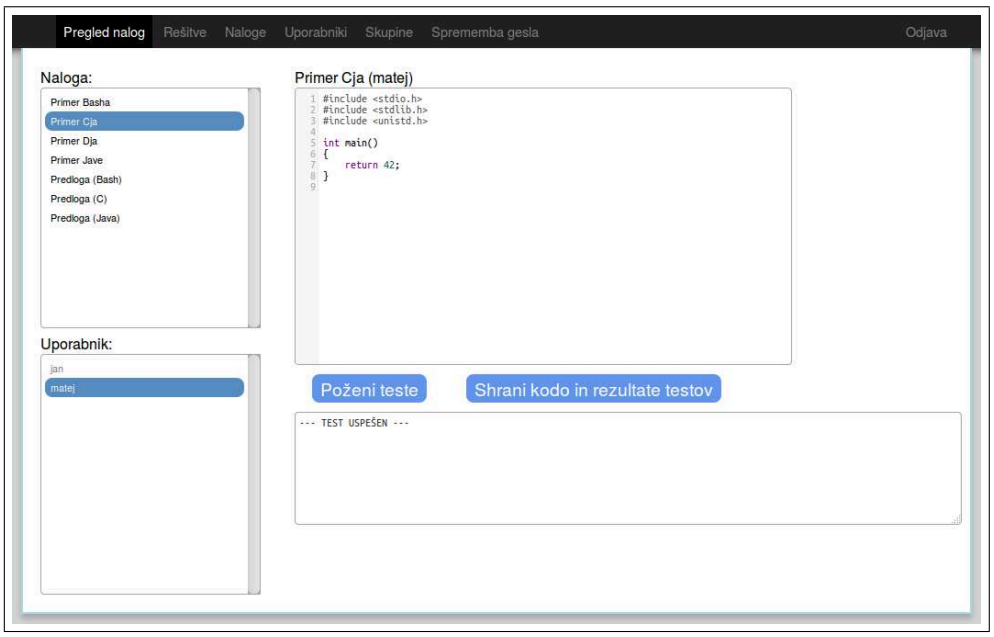


Slika 1: Oddaja naloge.

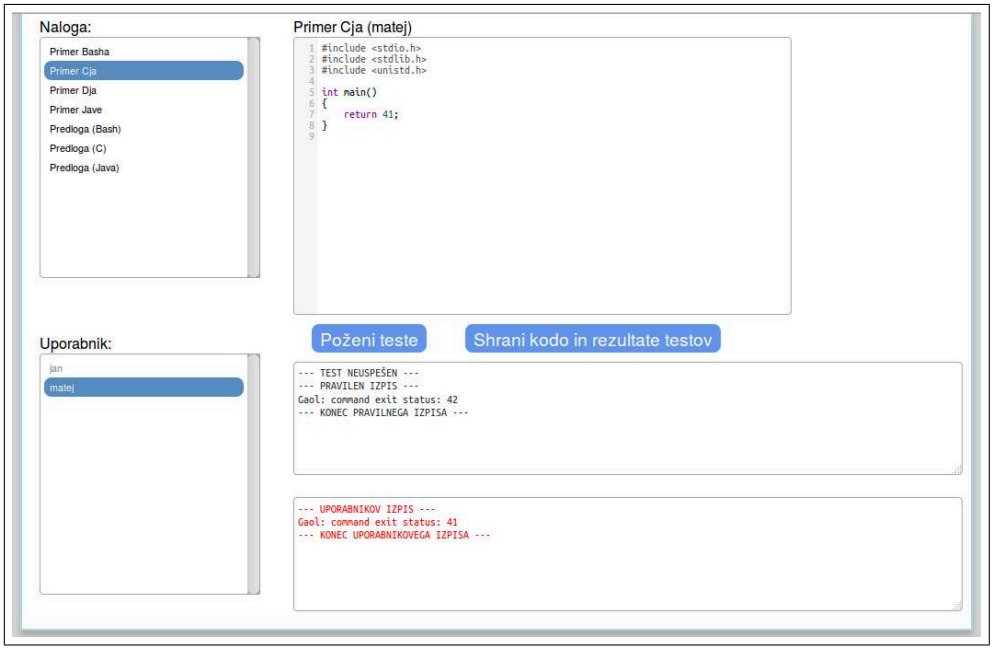
Skrbnikov pogled na naloge je drugačen. Pri pregledu nalog ima na voljo tako nalogo kot uporabnika, kjer vidi njegovo rešitev. Na levi strani okna, kjer izbere uporabnika, barva besedila označuje ali so bili uporabnikovi testi že uspešno pognani (zeleno) ali so bili pognani neuspešno (rdeče). Besedilo je lahko tudi sivo, če uporabnik naloge še ni reševal, ali črno, če je rešitev na voljo ampak še ni bila preizkušena. Ko teste požene sta možna dva izida, uspeh in neuspeh. V primeru uspeha (slika 2) se kot izpis programa izpiše samo `TEST USPEŠEN`, v primeru neuspeha (slika 3) pa imamo na voljo še izpis rešitve referenčnega programa in izpis uporabnikovega programa v rdeči pisavi.

Skrbnik ima tudi možnost posodobitve oddane rešitve naloge. To je uporabno v primeru, ko je napaka v programu trivialna in jo lahko takoj popravi. Primer takšne napake je preveč izpisanih novih vrstic ali kak podoben odvečen znak, zaradi katerega bi bil preizkus neuspešen.

Pri dodajanju in urejanju naloge imamo možnost novo nalogo dodati na osnovi druge naloge, ki v tem primeru služi kot predloga, lahko pa za urejanje izberemo obstoječo. Nastavitve naloge so razdeljene na tri dele (slika 4). Prvi vsebuje splošne podatke o nalogi: njen naziv, njeno aktivnost, ki vpliva na vidnost naloge običajnim uporabnikom, datum oddaje, do



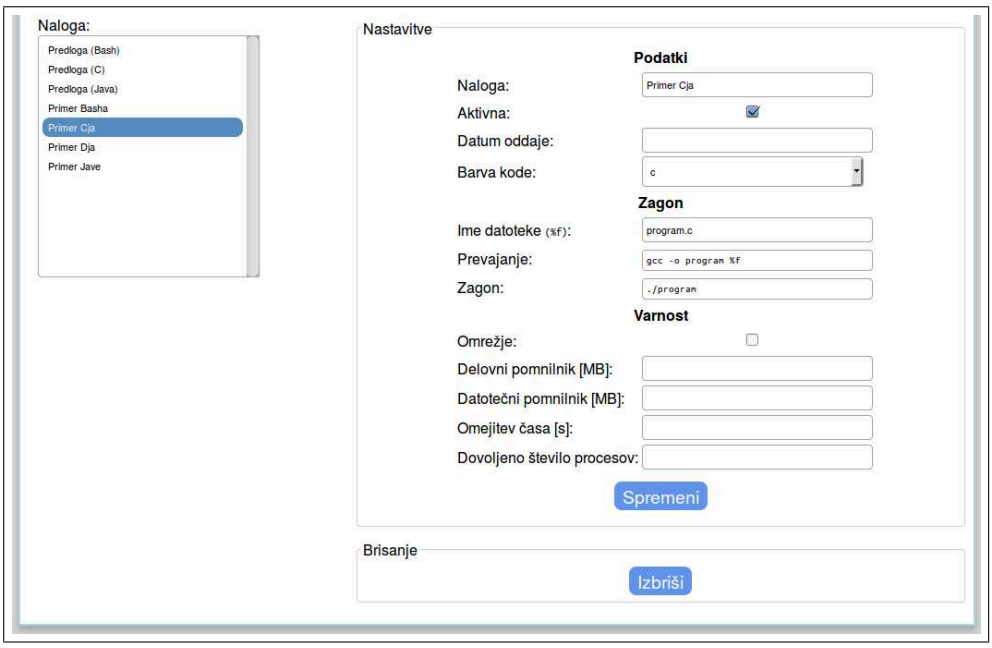
Slika 2: Uspešno rešena naloga.



Slika 3: Neuspešno rešena naloga.

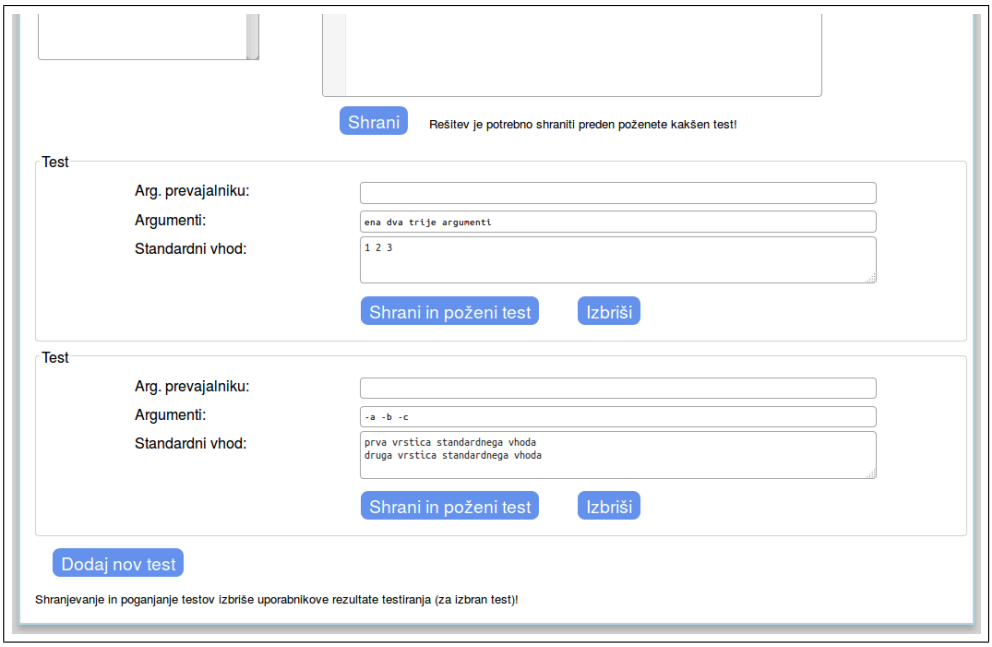
katerega je rešitev naloge dovoljeno spreminjati in izbira implementacije barvanja kode, ki jih vsebovana knjižnica za urejanje kode, `CodeMirror`, omogoča. Drugi del je vezan na poganjanje programov. Tam lahko določimo ime datoteke, v katero se kode shrani, ukaz za prevajalnik v ukazni vrstici in ukaz za zagon samega programa. Pri tem je vpis prevajalnika neobvezen, saj ga tolmačeni programski jeziki ne potrebujejo. Zadnji del so varnostne nastavitve. Če jih izpustimo se bodo uporabile privzete vrednosti, ki bi jih `gaol` uporabil, če v njegovi ukazni vrstici ne bi navedli argumentov. Sicer pa lahko programom omogočimo uporabo omrežja, lahko jim nastavimo omejitev uporabe delovnega pomnilnika, omejitev velikosti datotečnega sistema, časovno omejitev izvajanja in dovoljeno število procesov in niti, ki jih lahko ustvarijo.

Posamezni nalogi lahko v ločenem pogledu (zavihek Rešitve) vpišemo referenčni pro-



Slika 4: Dodajanje in nastavljanje nalog.

gram, ki predstavlja pravilno rešitev naloge. Pod kodo tega programa pa se nahajajo testni primeri (slika 5). Dodamo lahko poljubno število testov. Vsakemu lahko ločeno nastavimo zastavice prevajalnika, argumente programu z ukazne vrstice in vsebino standardnega vhoda programa. Ko test shranimo ga sistem izvede in izpiše standardni izhod programa, ki ga za nadaljnje primerjave shrani tudi v podatkovno bazo.



Slika 5: Dodajanje testov za nalogo.

Spletni sodniški sistem trenutno ni v redni uporabi, saj bi za to potreboval nekatere nadgradnje. Najprej bi bilo potrebno podpreti prijavo z uporabo OpenID ali identifikacijskega strežnika fakultete, saj je lokalno hranjenje gesel slaba praksa, hkrati pa zahteva dodaten trud s strani skrbnikov. Poleg tega bi bilo dobro podpreti prenos in urejanje večih datotek hkrati,

saj je to pri nekaterih programskih jezikih za zapletenejše programe, kot je Java, obvezno. Še ena omejitev sistema je samo en skrbniški uporabnik, »admin«, ki lahko vidi in počne vse. Razen teh pomanjkljivosti pa sistem dobro prikaže uporabnost varnega izvajanja programov v večji rešitvi in poziva k nadaljnjemu razvoju.

Poglavje 4

Sklepne ugotovitve

V okviru diplomskega dela smo raziskali mehanizme za omejevanje izvajanja programov v operacijskem sistemu GNU/Linux in ovrednotili njihovo uporabnost za rešitev težave diplomskega dela. Težava obsega varno izvedbo neznanih programov z namenom, da bi preizkusili njihovo delovanje brez nevarnosti poškodb našega sistema. Ker imamo namen preizkušati veliko programov, mora biti rešitev tudi učinkovita.

Posledica raziskave je razvoj programa `gaol`, ki omogoča varno in učinkovito izvajanje kateregakoli programa preko ukazne vrstice. Svoje cilje doseže z uporabo mehanizmov kot so imenski prostori (angl. *namespaces*) in nadzorne skupine (angl. *control groups*), ki neznan program izolirajo od sistemskih virov ostalih programov.

Prav tako smo razvili spletni sodniški sistem, ki `gaol` uporablja v ozadju. Z njim lahko učenci svoje programske rešitve naložijo na splet in preverijo njihove rezultate na testnih primerih. Zaradi visoke učinkovitosti sistema pa ga lahko uporabijo tudi za razvoj svojih nalog, saj so rezultati takojšnji.

V prihodnosti bo uporabnost programa `gaol` mogoče izboljšati s poenostavitvijo njegove namestitve. Trenutno je za to potrebnega kar nekaj znanja, saj so tehnologije relativno nove in se zato podpora med distribucijami operacijskega sistema GNU/Linux razlikuje. Poleg tega je še vedno mogoče izboljšati učinkovitost implementacije. Zdaj se za vzpostavitev varnega okolja uporablja programski paket LXC (*Linux containers*), ki bi ga lahko zaobšli in njegove ustrezne dele implementirali sami. Čeprav je `gaol` popolnoma delujoč pa se spletni sistem v živo ne uporablja, saj implementacija njegovega uporabniškega vmesnika ni izpopolnjena in potrebuje za praktično uporabo nadgradnje.

Literatura

[1] Kees Cook. user ns: arbitrary module loading, marec 2013. Dostopno na http://lwn.net/Articles/543539/.

[2] Jonathan Corbet. BPF: the universal in-kernel virtual machine, maj 2014. Dostopno na http://lwn.net/Articles/599755/.

[3] Jonathan Corbet. The unified control group hierarchy in 3.16, junij 2014. Dostopno na http://lwn.net/Articles/601840/.

[4] Jake Edge. Google's Chromium Sandbox, avgust 2009. Dostopno na http://lwn.net/Articles/347547.

[5] Jake Edge. Another LSM stacking approach, oktober 2012. Dostopno na http://lwn.net/Articles/518345/.

[6] Chris Evans. Chrome 20 on Linux and Flash sandboxing, julij 2012. Dostopno na http://scarybeastsecurity.blogspot.com/2012/07/chrome-20-on-linux-and-flash-sandboxing.html.

[7] Free Software Foundation. GNU core utilities manual — chroot invocation, 2014. Dostopno na http://www.gnu.org/software/coreutils/manual/coreutils.html#chroot-invocation.

[8] FreeBSD System Calls Manual. chroot(2), januar 2012. Dostopno na http://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2&manpath=FreeBSD+10.0-stable.

[9] FreeBSD System Calls Manual. ptrace(2), julij 2013. Dostopno na https://www.freebsd.org/cgi/man.cgi?query=ptrace&manpath=FreeBSD+10.1-stable.

[10] Aditya Kali. RFC: CGroup Namespaces, julij 2014. Dostopno na http://lwn.net/Articles/605903/.

[11] Max Kellerman. new cgroup controller "fork", november 2011. Dostopno na http://lwn.net/Articles/465574/.

[12] Michael Kerrisk. Anatomy of a user namespaces vulnerability, marec 2013. Dostopno na http://lwn.net/Articles/543273/.

[13] Michael Kerrisk. Namespaces in operation, part 4: more on PID namespaces, januar 2013. Dostopno na http://lwn.net/Articles/532748/.

[14] Sebastian Krahmer. CLONE_NEWUSER|CLONE_FS root exploit, marec 2013. Dostopno na http://lwn.net/Articles/543442/.

[15] Linux kernel documentation. Block IO Controller, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt.

[16] Linux kernel documentation. Cgroup unified hierarchy, april 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/unified-hierarchy.txt.

[17] Linux kernel documentation. CPU Accounting Controller, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/cpuacct.txt.

[18] Linux kernel documentation. Cpusets, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt.

[19] Linux kernel documentation. Device Whitelist Controller, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/devices.txt.

[20] Linux kernel documentation. Freezer subsystem, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt.

[21] Linux kernel documentation. HugeTLB Controller, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/hugetlb.txt.

[22] Linux kernel documentation. Memory Resource Controller, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/memory.txt.

[23] Linux kernel documentation. Network classifier cgroup, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/net_cls.txt.

[24] Linux kernel documentation. Network priority cgroup, december 2014. Dostopno na https://www.kernel.org/doc/Documentation/cgroups/net_prio.txt.

[25] Linux man-pages project. umask(2) - Linux manual page, januar 2008. Dostopno na http://man7.org/linux/man-pages/man2/umask.2.html.

[26] Linux man-pages project. chroot(2) - Linux manual page, september 2010. Dostopno na http://man7.org/linux/man-pages/man2/chroot.2.html.

[27] Linux man-pages project. fstab(5) - Linux manual page, avgust 2010. Dostopno na http://man7.org/linux/man-pages/man5/fstab.5.html.

[28] Linux man-pages project. ip(8) - Linux manual page, december 2011. Dostopno na http://man7.org/linux/man-pages/man8/ip.8.html.

[29] Linux man-pages project. setns(2) - Linux manual page, januar 2013. Dostopno na http://man7.org/linux/man-pages/man2/setns.2.html.

[30] Linux man-pages project. unshare(2) - Linux manual page, april 2013. Dostopno na http://man7.org/linux/man-pages/man2/unshare.2.html.

[31] Linux man-pages project. capabilities(7) - Linux manual page, september 2014. Dostopno na http://man7.org/linux/man-pages/man7/capabilities.7.html.

[32] Linux man-pages project. clone(2) - Linux manual page, avgust 2014. Dostopno na http://man7.org/linux/man-pages/man2/clone.2.html.

[33] Linux man-pages project. keyctl(2) - Linux manual page, januar 2014. Dostopno na http://man7.org/linux/man-pages/man2/keyctl.2.html.

[34] Linux man-pages project. mount(2) - Linux manual page, september 2014. Dostopno na http://man7.org/linux/man-pages/man2/mount.2.html.

[35] Linux man-pages project. mount(8) - Linux manual page, julij 2014. Dostopno na http://man7.org/linux/man-pages/man8/mount.8.html.

[36] Linux man-pages project. mq_overview(7) - Linux manual page, september 2014. Dostopno na http://man7.org/linux/man-pages/man7/mq_overview.7.html.

[37] Linux man-pages project. prctl(2) - Linux manual page, april 2014. Dostopno na http://man7.org/linux/man-pages/man2/prctl.2.html.

[38] Linux man-pages project. proc(5) - Linux manual page, julij 2014. Dostopno na http://man7.org/linux/man-pages/man5/proc.5.html.

[39] Linux man-pages project. svipc(7) - Linux manual page, september 2014. Dostopno na http://man7.org/linux/man-pages/man7/svipc.7.html.

[40] Andy Lutomirski in Eric W. Biederman. Re: CLONE_NEWUSER|CLONE_FS root exploit, marec 2013. Dostopno na http://lwn.net/Articles/543316/.

[41] Daniel Micay. GitHub repozitorij thestinger/playpen, 2013. Dostopno na https://github.com/thestinger/playpen.

[42] Todd C. Miller. Sudoers Manual, 2014. Dostopno na http://www.sudo.ws/sudo/sudoers.man.html.

[43] MozillaWiki. Security/Sandbox, avgust 2014. Dostopno na https://wiki.mozilla.org/Security/Sandbox.

[44] Lennart Poettering. mount-setup: change system mount propagation to shared by default, avgust 2012. Dostopno na http://cgit.freedesktop.org/systemd/systemd/commit/?id=b3ac5f8cb98757416d8660023d6564a7c411f0a0.

[45] The Chromium Team. setuid-sandbox, januar 2012. Dostopno na https://code.google.com/p/setuid-sandbox/.

[46] Evan Teran. Linux's ptrace API sucks!, januar 2008. Dostopno na http://blog.codef00.com/2008/01/29/linuxs-ptrace-api-sucks/.

[47] The ELF shell crew. Embedded ELF Debugging, januar 2005. Dostopno na http://phrack.org/issues/63/9.html.

[48] The Linux Foundation. bridge, november 2009. Dostopno na http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[49] Julien Tinnes. Introducing Chrome's next-generation Linux sandbox, september 2012. Dostopno na http://blog.cr0.org/2012/09/introducing-chromes-next-generation.html.

[50] Julien Tinnes in Chris Evans. Security In-Depth for Linux Software — Preventing and Mitigating Security Bugs, oktober 2009. Dostopno na https://www.cr0.org/paper/jt-ce-sid_linux.pdf.

[51] Guido Van 't Noordende, Ádám Balogh, Rutger Hofman, Frances M. T. Brazier, in Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. Objavljeno v *P*roc. 2nd International Conference on Security and Cryptography (SECRYPT), strani 414–423, julij 2007.